KHOA CÔNG NGHỆ

BÁO CÁO
BÀI TOÁN ĐỊNH TUYẾN SHIPPER

Người thực hiện: Sơn Tân

Khóa: 2022 – 2026

Giảng viên hướng dẫn: ThS. Nguyễn Quản Bá Hồng

Thành phố Hồ Chí Minh, tháng 6 năm 2025

Phần I: Mô hình Toán học Tổng quát

1. Giới thiệu

Bài toán định tuyến shipper (Vehicle Routing Problem – VRP) là một bài toán tối ưu hóa trong đó ta cần:

- Xác định một hoặc nhiều phương tiện (shipper/xe) xuất phát từ một điểm gốc (gọi là *depot*), đi qua tất cả các điểm khách hàng để giao hoặc nhận hàng.
- Mỗi điểm khách hàng phải được ghé đúng một lần.
- Tuân thủ các ràng buộc thực tế như:
  - Khả năng chứa tối đa của mỗi xe (*capacity*).
  - Khung giờ mà khách hàng chấp nhận giao nhận (*time windows*).
  - Thời gian phục vụ tại mỗi điểm (*service time*).
- Mục tiêu chung thường là *tối thiểu tổng chi phí* (có thể là quãng đường, thời gian hoặc chi phí tài chính kết hợp).

Dưới đây trình bày chi tiết mô hình *Linear Integer Programming* (MILP) tổng quát cho bài toán VRP có cả *time windows* và *capacity*. Phiên bản này đôi khi được gọi là *VRPTW-C* (Vehicle Routing Problem with Time Windows and Capacity).

2. Các tập và chỉ số

- $V = \{\, 0, 1, 2, \ldots, n \,\}$ là tập các *đỉnh* (điểm) trong đồ thị:
  - 0 là **depot** (kho trung tâm, nơi mỗi xe xuất phát và cuối cùng quay về).
  - $i = 1, 2, \ldots, n$ là các **điểm khách hàng** cần giao hoặc nhận hàng.
- $K = \{\, 1, 2, \ldots, m \,\}$ là tập các *xe* (shipper) có sẵn.
  - Nếu chỉ có một xe, thì $m = 1$ và $K = \{1\}$.
- Ký hiệu:

$$i,\, j \in V, \quad i \neq j; \qquad k \in K.$$

3. Tham số (Parameters)

$d_{ij} \in \mathbb{R}_{\geq 0}$: *chi phí* (khoảng cách hoặc chi phí vận chuyển) từ điểm i đến j. Thông thường $d_{ii} = 0$.

$\tau_{ij} \in \mathbb{R}_{\geq 0}$: *thời gian di chuyển* (phút hoặc giờ) từ điểm i đến j. Thông thường $\tau_{ij} = \frac{d_{ij}}{v}$ nếu có vận tốc v, hoặc lấy trực tiếp từ API giao thông.

$[e_i, l_i] \subset \mathbb{R}_{\geq 0}$: khung giờ (*time window*) tại điểm i, nghĩa là xe chỉ được phép bắt đầu phục vụ tại i trong khoảng

$$e_i \ \leq \ t_i \ \leq \ l_i.$$

Với depot ($i = 0$), ta có thể đặt $[e_0, l_0]$ là khung giờ làm việc của kho (ví dụ $[0, T_{\max}]$).

$s_i \in \mathbb{R}_{\geq 0}$: *thời gian phục vụ* (service time) tại điểm i. Sau khi xe đến i và bắt đầu phục vụ, nó phải dừng lại s_i đơn vị thời gian trước khi có thể rời đi. Thông thường $s_0 = 0$ tại depot.

$q_i \in \mathbb{R}_{\geq 0}$: *nhu cầu (demand)* tại điểm i, thể hiện khối lượng hoặc thể tích cần giao (hoặc nhận). Với $i = 0$ (depot), thường đặt $q_0 = 0$.

$Q_k \in \mathbb{R}_{>0}$: *capacity* (sức chứa tối đa) của xe k. Tổng nhu cầu trên một tuyến không được vượt quá Q_k.

$M \in \mathbb{R}_{>0}$: hằng số lớn (*big-M*) dùng trong ràng buộc logic để vô hiệu hoá khi biến nhị phân bằng 0. Chọn M đủ lớn để không vô tình cắt nhánh khả thi. Ví dụ:

$$M \ = \ T_{\max} + \max_{i,j \in V}\{\tau_{ij}\} + \max_{i \in V}\{s_i\},$$

hoặc $M = \sum_{i=1}^{n} q_i$ khi xét capacity.

4. Biến quyết định (Decision Variables)

$x_{ij}^k \in \{0, 1\}$:

$$x_{ij}^k = \begin{cases} 1, & \text{nếu xe } k \text{ đi trực tiếp từ } i \text{ sang } j, \\ 0, & \text{ngược lại.} \end{cases}$$

Với $i, j \in V,\ i \neq j,\ k \in K$.

- Nếu $x_{ij}^k = 1$, tuyến đường $(i \to j)$ thuộc tuyến của xe k.

$t_i^k \in \mathbb{R}_{\geq 0}$:

$$t_i^k = \text{thời điểm xe } k \text{ bắt đầu phục vụ tại điểm } i.$$

Với $i \in V,\ k \in K$.

- Phải thỏa $e_i \leq t_i^k \leq l_i$ nếu xe k đến đúng i. Nếu xe k không ghé i, t_i^k chỉ tồn tại khái niệm (có thể để tự động lớn).

$y_i^k \in \mathbb{R}_{\geq 0}$:

$$y_i^k = \text{tải trọng (load) còn lại trên xe } k \text{ ngay sau khi phục vụ tại } i.$$

Với $i \in V,\ k \in K$.

- Nếu xe k không ghé i, biến y_i^k không có ý nghĩa thực tế (có thể đặt $y_i^k = 0$).
- Khi bắt đầu tại depot 0, ta đặt $y_0^k = \sum_{i=1}^{n} q_i$ nếu xe xuất phát với toàn bộ hàng, hoặc gán $y_0^k \leq Q_k$ để linh hoạt hơn.

$u_i \in \mathbb{Z}$: chỉ dùng khi cần *loại chu trình con* (subtour elimination) trong trường hợp TSP (một xe, không có capacity/time windows chi tiết). Với VRP đa xe, ràng buộc luồng (flow) thường đã loại bỏ subtour, nên u_i không bắt buộc.

5. Hàm mục tiêu (Objective Function)

Giả sử mục tiêu là *tối thiểu tổng chi phí di chuyển*:

$$\min\ Z = \sum_{k \in K} \sum_{i \in V} \sum_{\substack{j \in V \\ j \neq i}} d_{ij}\, x_{ij}^k. \tag{OBJ}$$

- Nếu muốn thay bằng tối thiểu tổng thời gian, có thể ghi:

$$\min \sum_{k \in K} \sum_{i \in V} \sum_{j \in V,\, j \neq i} \tau_{ij}\, x_{ij}^k.$$

- Nếu muốn kết hợp đa tiêu chí, ví dụ cả chi phí d_{ij} và thời gian τ_{ij}, thì:

$$\min \sum_{k \in K} \sum_{i \in V} \sum_{j \in V,\, j \neq i} \left(\alpha\, d_{ij} \ +\ \beta\, \tau_{ij}\right) x_{ij}^k,$$

với α, β là các hệ số trọng số do người mô hình quy định.

6. Các ràng buộc (Constraints)

6.1. Ràng buộc mỗi khách được phục vụ đúng một lần

$$\forall\, i = 1, \ldots, n: \quad \sum_{k \in K} \sum_{\substack{j \in V \\ j \neq i}} x_{ij}^k \ =\ 1. \tag{C1}$$

- Diễn giải: mỗi điểm khách hàng $i \neq 0$ chỉ có đúng một cung $(i \to j)$ được chọn trong tất cả các xe. Khi đó, điểm i đã bị ghé đúng một lần.

6.2. Ràng buộc bảo toàn luồng cho mỗi xe

$$\forall\, k \in K,\ \forall\, h = 1, 2, \ldots, n: \quad \sum_{\substack{i \in V \\ i \neq h}} x_{ih}^k \ = \ \sum_{\substack{j \in V \\ j \neq h}} x_{hj}^k. \tag{C2}$$

$$\forall\, k \in K: \quad \sum_{\substack{j \in V \\ j \neq 0}} x_{0j}^k \ = \ 1, \quad \sum_{\substack{i \in V \\ i \neq 0}} x_{i0}^k \ = \ 1. \tag{C3}$$

- Với mỗi xe k và mỗi khách h, nếu xe k đến h (tồn tại $i \neq h$ sao cho $x_{ih}^k = 1$), thì phải có đúng một cung $x_{hj}^k = 1$ để xe rời h.
- Ràng buộc (C3) đảm bảo mỗi xe k *xuất phát* từ depot 0 đúng một lần và *quay về* depot 0 đúng một lần.

6.3. Ràng buộc Time Windows và Service Time

$$\forall\, i, j \in V,\ i \neq j,\ \forall\, k \in K: \quad t_j^k \ \geq \ t_i^k \ + \ s_i \ + \ \tau_{ij} \ - \ M\left(1 - x_{ij}^k\right). \tag{C4}$$

$$\forall\, i \in V,\ \forall\, k \in K: \quad e_i \ \leq \ t_i^k \ \leq \ l_i. \tag{C5}$$

- Nếu $x_{ij}^k = 1$, nghĩa là xe k đi từ i sang j. Khi đó, phải có

  $$t_j^k \ \geq \ t_i^k + s_i + \tau_{ij}.$$

  Đồng thời t_i^k phải nằm trong khung $[e_i,\, l_i]$, và t_j^k phải nằm trong $[e_j,\, l_j]$.

- Nếu $x_{ij}^k = 0$, ràng buộc (C4) trở thành

  $$t_j^k \ \geq \ t_i^k + s_i + \tau_{ij} - M,$$

  mà với M đủ lớn, t_j^k có thể là mọi giá trị thỏa (không ràng buộc thực tế).

6.4. Ràng buộc Capacity (Dung tích)

$$\forall\, i, j \in V,\ i \neq j,\ \forall\, k \in K: \quad y_j^k \ \geq \ y_i^k \ - \ q_j \ - \ M\left(1 - x_{ij}^k\right). \tag{C6}$$

$$\forall\, i \in V,\ \forall\, k \in K: \quad 0 \ \leq \ y_i^k \ \leq \ Q_k, \quad y_0^k \ = \ \sum_{i=1}^{n} q_i. \tag{C7}$$

- Giả sử mỗi xe k *xuất phát* từ depot 0 với tải trọng $y_0^k = \sum_{i=1}^n q_i$ (mọi hàng đều ban đầu nằm trên xe). Sau khi xe đến i và phục vụ, tải trọng còn lại giảm đi q_i.
- Nếu $x_{ij}^k = 1$, ta có

  $$y_j^k \ \geq \ y_i^k - q_j,$$

  nghĩa là sau khi phục vụ j, tải trọng mới trên xe là $y_i^k - q_j$.

- Ràng buộc $y_i^k \leq Q_k$ đảm bảo xe không chở quá sức chứa, và $y_i^k \geq 0$ đảm bảo không chở âm.

6.5. Ràng buộc loại trừ chu trình con (Subtour Elimination) – Chỉ dành cho TSP (một xe, không Capacity/Time Windows)

Khi chỉ có một xe ($m = 1$) và không xét y_i hay t_i phức tạp, để ngăn chặn chu trình tách biệt (subtour) không qua depot, ta dùng cách Held–Karp:

$$\forall\, i, j \in \{1, 2, \ldots, n\},\ i \neq j: \quad u_i - u_j + n\, x_{ij} \ \leq\ n - 1, \tag{C8}$$

$$\forall\, i = 1, 2, \ldots, n: \quad 1 \ \leq\ u_i \ \leq\ n, \quad u_i \in \mathbb{Z}. \tag{C9}$$

- Biến u_i là biến nguyên thể hiện *thứ tự* của điểm i trong chuỗi.
- Khi $x_{ij} = 1$, ràng buộc (C8) cho

$$u_i - u_j \ \leq\ n - 1 - n \ =\ -1 \quad \Longrightarrow \quad u_i + 1 \ \leq\ u_j,$$

  tức j phải đứng sau i.
- Nếu $x_{ij} = 0$, ràng buộc tự động lỏng nhờ $n\, x_{ij} = 0$.
- Nhờ (C8) và (C9), không thể tồn tại một chu trình con tách rời không qua depot.

6.6. Phạm vi biến (Variable Domains)

$$x_{ij}^k \in \{0, 1\}, \quad t_i^k \geq 0, \quad y_i^k \geq 0, \quad u_i \in \mathbb{Z}\,. \tag{C10}$$

- Với VRP đầy đủ ($m > 1$, xét cả t_i^k và y_i^k), giữ nguyên (C10).
- Nếu chỉ xét *TSP* (một xe, không capacity/time windows), có thể bỏ hoàn toàn y_i, y_0, t_i, và chỉ dùng x_{ij} kèm (C8),(C9).
- Nếu chỉ xét *TSPTW* (Time Windows, không Capacity), bỏ biến y_i và ràng buộc (C6),(C7).
- Nếu chỉ xét *CVRP* (capacity, không time windows), bỏ biến t_i và ràng buộc (C4),(C5).

7. Tóm tắt mô hình MILP

$$\min \quad \sum_{k\in K}\sum_{i\in V}\sum_{\substack{j\in V\\ j\neq i}} d_{ij}\, x_{ij}^{k} \quad \text{(OBJ)}$$

$$\text{s.t.} \quad (C1) \quad \forall\, i = 1,\dots,n: \ \sum_{k\in K}\sum_{\substack{j\in V\\ j\neq i}} x_{ij}^{k} = 1,$$

$$(C2) \quad \forall\, k\in K,\ \forall\, h = 1,\dots,n: \ \sum_{\substack{i\in V\\ i\neq h}} x_{ih}^{k} = \sum_{\substack{j\in V\\ j\neq h}} x_{hj}^{k},$$

$$(C3) \quad \forall\, k\in K: \ \sum_{\substack{j\in V\\ j\neq 0}} x_{0j}^{k} = 1, \quad \sum_{\substack{i\in V\\ i\neq 0}} x_{i0}^{k} = 1,$$

$$(C4) \quad \forall\, i,j\in V,\ i\neq j,\ k\in K: \ t_j^{k} \ \geq\ t_i^{k} + s_i + \tau_{ij} - M\,(1 - x_{ij}^{k}),$$

$$(C5) \quad \forall\, i\in V,\ k\in K: \ e_i \leq t_i^{k} \leq l_i,$$

$$(C6) \quad \forall\, i,j\in V,\ i\neq j,\ k\in K: \ y_j^{k} \ \geq\ y_i^{k} - q_j - M\,(1 - x_{ij}^{k}),$$

$$(C7) \quad \forall\, i\in V,\ k\in K: \ 0 \leq y_i^{k} \leq Q_k, \quad y_0^{k} = \sum_{i=1}^{n} q_i,$$

$$(C8) \quad \forall\, i,j\in\{1,\dots,n\},\ i\neq j: \ u_i - u_j + n\, x_{ij} \leq n-1,$$

$$(C9) \quad \forall\, i = 1,\dots,n: \ 1 \leq u_i \leq n, \quad u_i\in\mathbb{Z},$$

$$(C10) \quad x_{ij}^{k}\in\{0,1\},\ t_i^{k}\geq 0,\ y_i^{k}\geq 0,\ u_i\in\mathbb{Z}.$$

Chú ý:

- Nếu chỉ có $m = 1$ (một xe), ta có thể gán $x_{ij} \equiv x_{ij}^{1},\ t_i \equiv t_i^{1},\ y_i \equiv y_i^{1},\ Q \equiv Q_1$, và (C2),(C3) rút gọn tương ứng.
- Nếu không xét capacity, bỏ y_i^{k} và (C6),(C7). Nếu không xét time windows, bỏ t_i^{k} và (C4),(C5).
- Mô hình trên là dạng MILP; có thể dùng solver như Gurobi hoặc CPLEX để tìm nghiệm tối ưu khi n vừa phải ($n \leq 30$). Khi n lớn hơn, cần dùng các giải thuật heuristic hoặc meta-heuristic.

8. Ví dụ minh hoạ rút gọn (nếu chỉ một xe và có Time Windows)

Giả sử chỉ có một xe ($m = 1$), và ta xét bài toán TSPTW (Time Windows, không xét capacity chi tiết). Khi đó:

- $K = \{1\},\ x_{ij} \equiv x_{ij}^{1},\ t_i \equiv t_i^{1}$.
- Bỏ biến y_i, bỏ ràng buộc (C6),(C7).
- Bổ sung ràng buộc loại subtour (C8),(C9) để ngăn chu trình con.

Mô hình rút gọn trở thành:

$$
\begin{aligned}
\min \quad & \sum_{i=0}^{n} \sum_{\substack{j=0 \\ j \neq i}}^{n} d_{ij}\, x_{ij}, \\
\text{s.t.} \quad & (1)\ \forall\, i = 1, \ldots, n: \ \sum_{\substack{j=0 \\ j \neq i}}^{n} x_{ij} = 1, \quad \sum_{\substack{j=0 \\ j \neq i}}^{n} x_{ji} = 1, \\
& (2)\ \sum_{j=1}^{n} x_{0j} = 1, \quad \sum_{i=1}^{n} x_{i0} = 1, \\
& (3)\ \forall\, i, j \in V,\ i \neq j: \ \ t_j \geq t_i + s_i + \tau_{ij} - M\,(1 - x_{ij}), \\
& (4)\ \forall\, i \in V: \ \ e_i \leq t_i \leq l_i, \\
& (5)\ \forall\, i, j \in \{1, \ldots, n\},\ i \neq j: \ \ u_i - u_j + n\, x_{ij} \leq n - 1, \\
& (6)\ 1 \leq u_i \leq n,\ u_i \in \mathbb{Z}, \quad i = 1, \ldots, n, \\
& (7)\ x_{ij} \in \{0, 1\},\ t_i \geq 0.
\end{aligned}
$$

Đây chính là *TSP với Time Windows*, một trường hợp đặc biệt của VRP.

Kết luận Phần I: Trên đây là mô hình toán học tổng quát cho bài toán định tuyến shipper có cả khung giờ phục vụ và dung tích. Tùy từng trường hợp cụ thể (số xe, có/không capacity, có/không time windows), mô hình có thể được rút gọn hoặc mở rộng. Khi số điểm nhỏ, có thể giải MILP bằng solver; khi số điểm lớn, cần dùng giải thuật heuristic/meta-heuristic.

Phần II: Ví dụ Cụ thể

1. Dữ liệu ví dụ

Xét trường hợp **một shipper** (một xe) và **ba khách hàng** $1, 2, 3$. Depot ký hiệu là 0. Cho dữ liệu:

- **Tập điểm:**
$$V = \{0, 1, 2, 3\}.$$

- **Ma trận khoảng cách d_{ij} (km)**:
$$d = \begin{pmatrix} 0 & 10 & 12 & 20 \\ 10 & 0 & 8 & 15 \\ 12 & 8 & 0 & 18 \\ 20 & 15 & 18 & 0 \end{pmatrix}.$$

- **Thời gian di chuyển τ_{ij} (phút)**:
$$\tau_{ij} = 1.2 \times d_{ij}.$$

- **Thời gian phục vụ (Service Time)**:
$$s_0 = 0, \quad s_1 = 5, \quad s_2 = 5, \quad s_3 = 5 \quad \text{(phút)}.$$

- **Time Windows $[\, e_i,\, l_i \,]$**:
$$[\, e_0,\, l_0 \,] = [0, 100], \quad [\, e_1,\, l_1 \,] = [10, 50], \quad [\, e_2,\, l_2 \,] = [0, 40], \quad [\, e_3,\, l_3 \,] = [5, 60].$$

- **Nhu cầu (Demand) và Dung tích (Capacity)**:
$$q_1 = 3, \quad q_2 = 4, \quad q_3 = 2, \quad \sum_{i=1}^{3} q_i = 9, \quad Q = 10.$$

- **Hằng số Big–M**:
$$M = T_{\max} + \max_{i,j} \tau_{ij} + \max_{i} s_i = 100 + (1.2 \times 20) + 5 = 130.$$

2. Mô hình TSPTW (rút gọn)

Vì chỉ có một xe ($m = 1$), đặt:

$$x_{ij} \equiv x_{ij}^1, \quad t_i \equiv t_i^1, \quad u_i \equiv u_i, \quad Q \equiv 10.$$

Ràng buộc tổng demand $\sum_{i=1}^{3} q_i = 9 \leq 10$ đã thỏa, nên không cần xét chi tiết biến y_i.

$$\min \quad \sum_{i=0}^{3} \sum_{\substack{j=0 \\ j \neq i}}^{3} d_{ij}\, x_{ij}$$

$$\text{s.t.} \quad (1)\ \forall\, i = 1,2,3: \quad \sum_{\substack{j=0 \\ j \neq i}}^{3} x_{ij} = 1, \quad \sum_{\substack{j=0 \\ j \neq i}}^{3} x_{ji} = 1,$$

$$(2)\ \sum_{j=1}^{3} x_{0j} = 1, \quad \sum_{i=1}^{3} x_{i0} = 1,$$

$$(3)\ \forall\, i, j \in \{0,1,2,3\},\ i \neq j: \quad t_j \ \geq\ t_i + s_i + \tau_{ij} \ -\ 130\,(1 - x_{ij}),$$

$$(4)\ \forall\, i = 0,1,2,3: \quad e_i \ \leq\ t_i \ \leq\ l_i,$$

$$(5)\ \forall\, i, j \in \{1,2,3\},\ i \neq j: \quad u_i - u_j + 3\, x_{ij} \leq 2,$$

$$(6)\ 1 \leq u_i \leq 3, \quad u_i \in \mathbb{Z}, \quad i = 1,2,3,$$

$$(7)\ x_{ij} \in \{0,1\}, \quad t_i \geq 0.$$

3. Liệt kê và Kiểm tra các tuyến

Với $n = 3$, có $3! = 6$ hoán vị của ba khách $\{1,2,3\}$. Mỗi hoán vị tương ứng một chuỗi khởi đầu tại 0, ghé qua $\{1,2,3\}$, rồi quay về 0. Cụ thể:

$$\text{(a)}\ 0 \to 1 \to 2 \to 3 \to 0,$$
$$\text{(b)}\ 0 \to 1 \to 3 \to 2 \to 0,$$
$$\text{(c)}\ 0 \to 2 \to 1 \to 3 \to 0,$$
$$\text{(d)}\ 0 \to 2 \to 3 \to 1 \to 0,$$
$$\text{(e)}\ 0 \to 3 \to 1 \to 2 \to 0,$$
$$\text{(f)}\ 0 \to 3 \to 2 \to 1 \to 0.$$

Với mỗi tuyến, ta kiểm tra theo các bước:

1. **Tính tổng chi phí (distance)**:

$$\text{Cost} = \sum_{\text{các cạnh } (i \to j)} d_{ij}.$$

2. **Tính thời gian di chuyển và phục vụ**:

$$\tau_{ij} = 1.2\, d_{ij}, \quad t_{\text{đến } j} = \max\big(e_j,\ t_i + s_i + \tau_{ij}\big).$$

   Nếu $t_j > l_j$, tuyến không khả thi.

3. **Quay về depot** sau khi phục vụ điểm cuối:

$$t_{\text{cuối}} + \tau_{\text{cuối}\to 0}.$$

(a) $0 \to 1 \to 2 \to 3 \to 0$

- $0 \to 1$:

$$\tau_{01} = 1.2 \times 10 = 12, \quad t_1 = \max(10,\ 0 + 12) = 12, \quad t_{\text{kết thúc}_1} = 12 + 5 = 17.$$

- $1 \to 2$:

$$\tau_{12} = 1.2 \times 8 = 9.6, \quad t_2 = \max(0,\ 17 + 9.6) = 26.6, \quad t_{\text{kết thúc}_2} = 26.6 + 5 = 31.6.$$

- $2 \to 3$:

$$\tau_{23} = 1.2{\times}18 = 21.6, \quad t_3 = \max(5,\ 31.6{+}21.6) = 53.2, \quad t_{\text{kết thúc}_3} = 53.2{+}5 = 58.2.$$

- $3 \to 0$:

$$\tau_{30} = 1.2 \times 20 = 24, \quad t_{\text{về depot}} = 58.2 + 24 = 82.2.$$

Kiểm tra Time Windows:

$$t_1 = 12 \le 50, \quad t_2 = 26.6 \le 40, \quad t_3 = 53.2 \le 60.$$

Tất cả thỏa. *Tổng khoảng cách:*

$$10 + 8 + 18 + 20 = 56.$$

$\Rightarrow$ (a) *Khả thi*, Chi phí = 56.

(b) $0 \to 1 \to 3 \to 2 \to 0$

- $0 \to 1$: $t_1 = 12$, kết thúc tại 17.
- $1 \to 3$:

$$\tau_{13} = 1.2 \times 15 = 18, \quad t_3 = \max(5,\ 17 + 18) = 35, \quad t_{\text{kết thúc}_3} = 35 + 5 = 40.$$

- $3 \to 2$:

$$\tau_{32} = 1.2 \times 18 = 21.6, \quad t_2 = \max(0,\ 40 + 21.6) = 61.6.$$

Tuy nhiên $l_2 = 40$, nên $t_2 = 61.6 > 40 \Rightarrow$ *không khả thi.*

(c) $0 \to 2 \to 1 \to 3 \to 0$

- $0 \to 2$:

$$\tau_{02} = 1.2 \times 12 = 14.4, \quad t_2 = \max(0,\ 0 + 14.4) = 14.4, \quad t_{\text{kết thúc}_2} = 14.4 + 5 = 19.4.$$

- $2 \to 1$:

$$\tau_{21} = 1.2 \times 8 = 9.6, \quad t_1 = \max(10,\ 19.4 + 9.6) = 29.0, \quad t_{\text{kết thúc}_1} = 29.0 + 5 = 34.0.$$

- $1 \to 3$:

$$\tau_{13} = 18, \quad t_3 = \max(5,\ 34.0 + 18) = 52.0, \quad t_{\text{kết thúc}_3} = 52.0 + 5 = 57.0.$$

- $3 \to 0$:

$$\tau_{30} = 24, \quad t_{\text{về depot}} = 57.0 + 24 = 81.0.$$

Kiểm tra Time Windows:

$$t_2 = 14.4 \le 40, \quad t_1 = 29.0 \le 50, \quad t_3 = 52.0 \le 60.$$

Tất cả thỏa. *Tổng khoảng cách:*

$$12 + 8 + 15 + 20 = 55.$$

$\Rightarrow$ (c) *Khả thi*, Chi phí = 55.

(d) $0 \to 2 \to 3 \to 1 \to 0$

- $0 \to 2$: $t_2 = 14.4,\ t_{\text{kết thúc}_2} = 19.4.$

- $2 \to 3$:

$$\tau_{23} = 21.6, \quad t_3 = \max(5,\ 19.4 + 21.6) = 41.0, \quad t_{\text{kết thúc}_3} = 41.0 + 5 = 46.0.$$

- $3 \to 1$:

$$\tau_{31} = 1.2 \times 15 = 18, \quad t_1 = \max(10,\ 46.0 + 18) = 64.0.$$

Tuy nhiên $l_1 = 50$, nên $t_1 = 64.0 > 50 \Rightarrow$ *không khả thi.*

(e) $0 \to 3 \to 1 \to 2 \to 0$

- $0 \to 3$:

$$\tau_{03} = 1.2 \times 20 = 24, \quad t_3 = \max(5,\ 0 + 24) = 24, \quad t_{\text{kết thúc}_3} = 29.$$

- $3 \to 1$:

$$\tau_{31} = 18, \quad t_1 = \max(10,\ 29 + 18) = 47, \quad t_{\text{kết thúc}_1} = 52.$$

- $1 \to 2$:

$$\tau_{12} = 9.6, \quad t_2 = \max(0,\ 52 + 9.6) = 61.6.$$

Nhưng $l_2 = 40$, nên $t_2 = 61.6 > 40 \Rightarrow$ *không khả thi.*

(f) $0 \to 3 \to 2 \to 1 \to 0$

- $0 \to 3$: $t_3 = 24,\ t_{\text{kết thúc}_3} = 29.$

- $3 \to 2$:

$$\tau_{32} = 21.6, \quad t_2 = \max(0,\ 29 + 21.6) = 50.6.$$

Nhưng $l_2 = 40$, nên $t_2 = 50.6 > 40 \Rightarrow$ *không khả thi.*

4. Kết quả và Lựa chọn Tối ưu

Chỉ có hai tuyến khả thi:

$$\begin{aligned} &\text{(a) } 0 \to 1 \to 2 \to 3 \to 0, \quad \text{Chi phí} = 56, \\ &\text{(c) } 0 \to 2 \to 1 \to 3 \to 0, \quad \text{Chi phí} = 55. \end{aligned}$$

Tuyến (c) là **tuyến tối ưu** với tổng khoảng cách $\boxed{55}$. Cụ thể:

$$0 \ \to\ 2 \ \to\ 1 \ \to\ 3 \ \to\ 0,$$

với:

$$\begin{cases} t_0 = 0, \\ t_2 = 14.4 \quad (\leq 40), \\ t_1 = 29.0 \quad (\leq 50), \\ t_3 = 52.0 \quad (\leq 60), \\ t_{\text{về depot}} = 81.0. \end{cases}$$

Kết luận chung

- Kết quả: tuyến tối ưu $0 \to 2 \to 1 \to 3 \to 0$ với chi phí 55 và thỏa Time Windows, Capacity.
- Khi n lớn, cần sử dụng solver MILP hoặc các giải thuật heuristic/meta-heuristic để giải bài toán trong thời gian hợp lý.

Phần III: Dựng thuật toán

3.1 Tổng quan chiến lược giải quyết

Để giải bài toán định tuyến shipper (TSPTW/VRPTW) đã mô hình hóa ở Phần I, ta có thể chọn hai hướng chính:

- **Giải thuật chính xác (Exact)**: Áp dụng *Branch & Bound* hoặc *Dynamic Programming* (Held–Karp mở rộng cho Time Windows) để tìm lời giải tối ưu tuyệt đối. Chỉ khả thi khi số điểm n nhỏ (thông thường $n \leq 15$).
- **Giải thuật gần đúng (Heuristic/Meta-heuristic)**: Khi n lớn (≥ 20), sử dụng các phương pháp như *Nearest Neighbor*, *Insertion Heuristic*, *2-opt / 3-opt*, *Tabu Search*, *Simulated Annealing*, *Genetic Algorithm*, *Ant Colony Optimization*,... để tìm nghiệm gần tối ưu trong thời gian chấp nhận được.

Trong báo cáo này, phần tiếp theo sẽ mô tả chi tiết:

- Một ví dụ triển khai **Branch & Bound** mở rộng cho TSPTW (với một xe).
- Một giải thuật **Heuristic cơ bản** (sinh lời giải khởi tạo bằng *Nearest Neighbor*, sau đó cải tiến bằng *2-opt* kèm kiểm tra Time Windows).

3.2 Giải thuật chính xác: Branch & Bound cho TSPTW (một xe)

3.2.1 Ý tưởng chính

1. Sử dụng *Branch & Bound* trên không gian hoán vị của n khách. Mỗi nút trong cây tìm kiếm tương ứng với một *partial route* (chuỗi đã ghé một số điểm).
2. Tại mỗi nút, lưu trữ hai thông tin:
   - **Route partial** $R = (0 = i_0, i_1, \ldots, i_k)$ – các đỉnh đã ghé.
   - **Thời gian hiện tại** t_k khi đã phục vụ i_k.
3. Từ nút này, mở rộng sang các đỉnh j chưa ghé, tạo nút con $R' = (i_0, i_1, \ldots, i_k, j)$ nếu
$$t_k + s_{i_k} + \tau_{i_k j} \leq l_j.$$
   Tính thời gian bắt đầu tại j:
$$t_j = \max\big(e_j,\ t_k + s_{i_k} + \tau_{i_k j}\big).$$

4. Khi $k = n$ (đã ghé hết n khách), thêm đỉnh quay về depot 0 nếu

$$t_n + s_{i_n} + \tau_{i_n 0} \leq l_0.$$

   Khi đó ta có một route đầy đủ.

5. Sử dụng **Bounding** (ước lượng tối thiểu chi phí còn lại) để cắt nhánh:

$$\text{LowerBound} = \text{Cost}(R) + \sum_{j \in U} \min_{x \in V} d_{jx},$$

   trong đó U là tập các đỉnh chưa ghé, và $\min_{x \in V} d_{jx}$ là ước tính chi phí nhỏ nhất để ghé j (bỏ qua Time Windows).

6. Nếu LowerBound $\geq$ BestCostFound, cắt bỏ nhánh này.

3.2.2 Pseudocode

```
Input:   n điểm khách {1,...,n}, ma trận d_ij, tau_ij,
         Time Windows [e_i,l_i], Service Time s_i.

Output:  Route tối ưu (0->...->0) với chi phí MinCost.

Initialize:
  BestRoute <- null
  BestCost  <- +INFINITY

Procedure BranchAndBound_Partial(R, t_R, cost_R, U):
  // R = [0 = i_0, i_1, ..., i_k]: partial route
  // t_R = thời gian hiện tại khi đã phục vụ i_k
  // cost_R = chi phí đã đi của R
  // U = tập đỉnh chưa ghé

  if U is empty then
    // Đã ghé hết, kiểm tra quay về depot
    i_last <- R[k]
    if t_R + s[i_last] + tau[i_last][0] <= l[0] then
      cost_full <- cost_R + d[i_last][0]
      if cost_full < BestCost then
        BestCost  <- cost_full
        BestRoute <- R + [0]
    end if
```

```
    return
  end if

  // Tính LowerBound cho nhánh hiện tại
  LB <- cost_R + lower_bound_estimate(U)
  if LB >= BestCost then
    return  // Cắt nhánh
  end if

  // Mở rộng nhánh: thử ghé từng j in U
  for each j in U do
    arrival_j <- max(e[j], t_R + s[R[k]] + tau[R[k]][j])
    if arrival_j <= l[j] then
      new_cost <- cost_R + d[R[k]][j]
      R' <- R + [j]
      t_R' <- arrival_j
      U' <- U without j
      BranchAndBound_Partial(R', t_R', new_cost, U')
    end if
  end for
end Procedure

Main:
  R0    <- [0]        // Bắt đầu tại depot
  t0    <- 0
  cost0 <- 0
  U0    <- {1,2,...,n}
  BranchAndBound_Partial(R0, t0, cost0, U0)

Return BestRoute, BestCost
```

Chú thích:

- `lower_bound_estimate(U)` có thể được cài đơn giản là tổng các $\min_x d_{jx}$ với $j \in U$, hoặc phức tạp hơn bằng giải bài toán *minimum spanning tree* trên $U \cup \{0\}$.
- Cắt nhánh (pruning) dựa vào LB $\geq$ BestCost giúp giảm đáng kể không gian tìm kiếm.
- Thuật toán này phù hợp cho $n \leq 15$. Khi $n > 15$, thời gian tính toán tăng theo hàm mũ; không khả thi thực tế.

3.3 Giải thuật gần đúng: Nearest Neighbor + 2-opt cho TSPTW

3.3.1 Ý tưởng chính Đối với n lớn, để tìm nghiệm gần tối ưu trong thời gian ngắn, ta có thể:

1. **Sinh lời giải khởi tạo (Initial Solution)** bằng *Nearest Neighbor có Time Windows*:
   - Bắt đầu từ depot 0, tại mỗi bước, chọn đỉnh $j \in U$ sao cho khoảng cách $d_{i,j}$ nhỏ nhất và
   $$\max\big(e_j,\ t_i + s_i + \tau_{i,j}\big) \le l_j.$$
   - Cập nhật thời gian t_j và tập $U \leftarrow U \setminus \{j\}$. Nếu không có đỉnh nào thỏa, dừng vòng lặp.
2. **Cải tiến (Local Search)** bằng *2-opt* mở rộng cho Time Windows:
   - Lặp đến khi không còn cải thiện: Duyệt cặp cạnh $\big((i_{p-1} \to i_p), (i_q \to i_{q+1})\big)$, hoán đổi để thành $(i_{p-1} \to i_q), (i_p \to i_{q+1})$.
   - Sau khi hoán đổi, kiểm tra lại toàn bộ thời gian đến từng điểm để đảm bảo Time Windows. Nếu khả thi và giảm chi phí, chấp nhận đổi.

3.3.2 Pseudocode

```
Input:   n điểm khách {1,...,n}, ma trận d_ij, tau_ij,
         Time Windows [e_i,l_i], Service Time s_i.

Output:  Một route khả thi gần tối ưu và chi phí tương ứng.

1. // Khởi tạo bằng Nearest Neighbor có Time Windows
   route       <- [0]
   t_current   <- 0
   U           <- {1,2,...,n}
   while U is not empty do
     i <- last element of route
     feasible_set <- empty
     for each j in U do
       arrival_j <- max(e[j], t_current + s[i] + tau[i][j])
       if arrival_j <= l[j] then
         add (j, arrival_j) to feasible_set
       end if
     end for
     if feasible_set is empty then
```

```
      break   // Không còn khách nào ghé được → kết thúc
    end if
    // Chọn j_min sao cho d[i][j_min] nhỏ nhất trong feasible_set
    j_min         <- argmin_{(j,_) in feasible_set} d[i][j]
    arrival_j_min <- corresponding arrival time
    append j_min to route
    t_current    <- arrival_j_min
    remove j_min from U
  end while
  append 0 to route  // Quay về depot

2. // Tính chi phí ban đầu
  cost <- 0
  for k = 1 to length(route)-1 do
    cost <- cost + d[route[k-1]][route[k]]
  end for

3. // Cải tiến bằng 2-opt có xét Time Windows
  improved <- true
  while improved do
    improved <- false
    for p = 1 to length(route)-3 do
      for q = p+1 to length(route)-2 do
        // Thử hoán đổi đoạn [p..q]
        new_route <- two_opt_swap(route, p, q)
        if check_time_windows(new_route) then
          new_cost <- compute_cost(new_route)
          if new_cost < cost then
            route <- new_route
            cost  <- new_cost
            improved <- true
            break out of both loops
          end if
        end if
      end for
      if improved then break end if
    end for
  end while
```

```
Return route, cost

// Hàm phụ trợ:
function two_opt_swap(route, p, q):
  // Trả về route mới:
  // giữ nguyên route[0..p-1], đảo ngược route[p..q], giữ route[q+1..end]
  new_route <- route[0..p-1]
  append reversed(route[p..q]) to new_route
  append route[q+1..end] to new_route
  return new_route

function check_time_windows(route):
  t <- 0
  for k = 1 to length(route)-1 do
    i <- route[k-1]
    j <- route[k]
    t_arrive <- max(e[j], t + s[i] + tau[i][j])
    if t_arrive > l[j] then
      return false
    end if
    t <- t_arrive
  end for
  return true

function compute_cost(route):
  sum <- 0
  for k = 1 to length(route)-1 do
    sum <- sum + d[route[k-1]][route[k]]
  end for
  return sum
```

Chú thích:

- Bước 1 (Nearest Neighbor có Time Windows) đảm bảo route khởi tạo *khả thi* (nếu có thể ghé hết). Nếu không thể ghé hết, kết thúc vòng lặp sớm.
- Bước 3 (2-opt có kiểm tra Time Windows) lặp đến khi không còn hoán đổi nào cải thiện chi phí. Mỗi khi hoán đổi, cần duyệt lại toàn bộ route để tính thời gian đến từng điểm và kiểm tra Time Windows.
- Độ phức tạp của thuật toán:

  - Nearest Neighbor: $O(n^2)$.
  - 2-opt: Mỗi lần xét cặp (p, q) cần $O(n)$ để kiểm tra Time Windows; có $O(n^2)$ cặp, và nếu lặp L lần, tổng $O(L \cdot n^3)$. Thực tế L thường nhỏ hơn n.

- Phương pháp này chỉ tìm được nghiệm gần tối ưu nhưng chạy nhanh với n lớn (khoảng 50–200).

3.4 Lựa chọn và so sánh

- Nếu số khách $n \leq 15$, **Branch & Bound** (Phần 3.2) cho kết quả tối ưu tuyệt đối. Tuy nhiên khi $n > 15$, thời gian tính toán tăng theo hàm mũ; không khả thi thực tế.

- Khi n lớn (≥ 20), nên dùng **Nearest Neighbor + 2-opt** (Phần 3.3). Để cải thiện chất lượng, có thể mở rộng thêm:
  - *Insertion Heuristic* (chèn theo chi phí tăng nhỏ nhất thoả Time Windows).
  - *Meta-heuristic* (Simulated Annealing, Tabu Search, Genetic Algorithm) để thoát khỏi cực tiểu cục bộ.

- Trong thực nghiệm, có thể so sánh:

  $$\text{Chi phí của Branch \& Bound} \quad \text{và} \quad \text{Chi phí của Heuristic},$$

  cùng với *thời gian chạy* để chọn thuật toán phù hợp với quy mô và yêu cầu ứng dụng.

Kết luận Phần III:
Đã trình bày chi tiết hai lớp thuật toán: **chính xác** (Branch & Bound cho TSPTW) và **gần đúng** (Nearest Neighbor + 2-opt). Tùy theo số lượng điểm và yêu cầu thời gian chạy, người triển khai có thể lựa chọn hoặc kết hợp hai cách trên để giải bài toán định tuyến shipper hiệu quả nhất.

Phần IV: Triển khai và Kiểm thử

```
// File: vrptw_general.cpp
#include <bits/stdc++.h>
using namespace std;

static const double INF = 1e18;

int n;
vector<vector<int>> d;
vector<vector<double>> tau;
vector<double> e_win, l_win, s_time;
double M;
double bestCostBB = INF;
vector<int> bestRouteBB;

double lower_bound_estimate(const vector<int>& U) {
    double sum = 0;
    for (int j : U) {
        int mn = INT_MAX;
        for (int x = 0; x <= n; x++) {
            if (x == j) continue;
            mn = min(mn, d[j][x]);
        }
        sum += mn;
    }
    return sum;
}

void BB_recursive(vector<int>& route, double t_cur, double
   cost_cur, vector<int>& U) {
    if (U.empty()) {
        int last = route.back();
        double arr0 = max(e_win[0], t_cur + s_time[last] + tau[
           last][0]);
        if (arr0 <= l_win[0]) {
            double totalCost = cost_cur + d[last][0];
            if (totalCost < bestCostBB) {
                bestCostBB = totalCost;
                bestRouteBB = route;
                bestRouteBB.push_back(0);
```

```
            }
        }
        return;
    }
    double LB = cost_cur + lower_bound_estimate(U);
    if (LB >= bestCostBB) return;
    int last = route.back();
    for (int idx = 0; idx < (int)U.size(); idx++) {
        int j = U[idx];
        double arrival = max(e_win[j], t_cur + s_time[last] + tau[
           last][j]);
        if (arrival <= l_win[j]) {
            double newCost = cost_cur + d[last][j];
            route.push_back(j);
            double hold_t = t_cur;
            t_cur = arrival;
            int saved = U[idx];
            U.erase(U.begin() + idx);
            BB_recursive(route, t_cur, newCost, U);
            U.insert(U.begin() + idx, saved);
            route.pop_back();
            t_cur = hold_t;
        }
    }
}

void solveBranchAndBound() {
    vector<int> route = {0};
    double t0 = 0, cost0 = 0;
    vector<int> U;
    for (int i = 1; i <= n; i++) U.push_back(i);
    BB_recursive(route, t0, cost0, U);
}

double compute_cost(const vector<int>& route) {
    double sum = 0;
    for (int i = 1; i < (int)route.size(); i++) {
        sum += d[route[i-1]][route[i]];
    }
    return sum;
}
```

```
bool check_time_windows(const vector<int>& route) {
    double t = 0;
    for (int i = 1; i < (int)route.size(); i++) {
        int u = route[i-1], v = route[i];
        double arrival = max(e_win[v], t + s_time[u] + tau[u][v]);
        if (arrival > l_win[v]) return false;
        t = arrival;
    }
    return true;
}

vector<int> two_opt_swap(const vector<int>& route, int p, int q) {
    vector<int> newr;
    for (int i = 0; i < p; i++) newr.push_back(route[i]);
    for (int i = q; i >= p; i--) newr.push_back(route[i]);
    for (int i = q+1; i < (int)route.size(); i++) newr.push_back(
       route[i]);
    return newr;
}

vector<int> nearest_neighbor_init() {
    vector<int> route = {0};
    vector<bool> used(n+1, false);
    used[0] = true;
    double tcur = 0;
    int cur = 0;
    for (int step = 1; step <= n; step++) {
        int nxt = -1;
        double bd = INF, ba = INF;
        for (int j = 1; j <= n; j++) {
            if (used[j]) continue;
            double arrival = max(e_win[j], tcur + s_time[cur] +
               tau[cur][j]);
            if (arrival <= l_win[j] && d[cur][j] < bd) {
                bd = d[cur][j];
                ba = arrival;
                nxt = j;
            }
        }
        if (nxt == -1) break;
```

```
        route.push_back(nxt);
        used[nxt] = true;
        tcur = ba;
        cur = nxt;
    }
    route.push_back(0);
    return route;
}

void two_opt_improve(vector<int>& route) {
    double bestC = compute_cost(route);
    bool improved = true;
    while (improved) {
        improved = false;
        int sz = (int)route.size();
        for (int p = 1; p < sz - 2; p++) {
            for (int q = p + 1; q < sz - 1; q++) {
                vector<int> cand = two_opt_swap(route, p, q);
                if (!check_time_windows(cand)) continue;
                double cc = compute_cost(cand);
                if (cc < bestC) {
                    bestC = cc;
                    route = move(cand);
                    improved = true;
                    break;
                }
            }
            if (improved) break;
        }
    }
}

void solveHeuristic(vector<int>& bestRouteH, double& bestCostH) {
    vector<int> route = nearest_neighbor_init();
    if ((int)route.size() == n+2) {
        two_opt_improve(route);
    }
    bestRouteH = route;
    bestCostH = compute_cost(route);
}
```

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    cin >> n;
    d.assign(n+1, vector<int>(n+1));
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            cin >> d[i][j];
        }
    }
    e_win.assign(n+1, 0);
    l_win.assign(n+1, 0);
    s_time.assign(n+1, 0);
    for (int i = 0; i <= n; i++) {
        cin >> e_win[i] >> l_win[i] >> s_time[i];
    }

    tau.assign(n+1, vector<double>(n+1, 0));
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            tau[i][j] = 1.2 * d[i][j];
        }
    }
    double maxTau = 0, maxS = 0;
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            maxTau = max(maxTau, tau[i][j]);
        }
        maxS = max(maxS, s_time[i]);
    }
    M = l_win[0] + maxTau + maxS;

    solveBranchAndBound();
    cout << "===␣Branch␣&␣Bound␣(Exact)␣===\n";
    if (bestCostBB >= INF/2) {
        cout << "Khong␣tim␣duoc␣lich␣trinh␣kha␣thi\n";
    } else {
        cout << "Chi␣phi␣toi␣uu:␣" << bestCostBB << "\n";
        cout << "Lich␣trinh:␣";
        for (int v : bestRouteBB) cout << v << "␣";
```

```
        cout << "\n";
    }

    vector<int> bestRouteH;
    double bestCostH;
    solveHeuristic(bestRouteH, bestCostH);
    cout << "\n=== Heuristic (Nearest Neighbor + 2-opt) ===\n";
    if ((int)bestRouteH.size() < n+2) {
        cout << "Heuristic khong ghe du " << n << " khach\n";
        cout << "Lich trinh hien tai: ";
        for (int v : bestRouteH) cout << v << " ";
        cout << "\n";
    } else {
        cout << "Chi phi tim duoc: " << bestCostH << "\n";
        cout << "Lich trinh: ";
        for (int v : bestRouteH) cout << v << " ";
        cout << "\n";
    }

    return 0;
}
```

Giải thích

Phần mã này cài đặt hai phương pháp để giải bài toán TSPTW tổng quát với một xe:

Đầu tiên là phương pháp Branch & Bound (Exact) nhằm tìm lời giải tối ưu tuyệt đối. Khi số khách n nhỏ (thí dụ $n \leq 10$), thuật toán sẽ liệt kê dần các chuỗi hành trình khả thi, sử dụng ước lượng "lower bound" để cắt nhánh và bỏ các nhánh không thể cải thiện chi phí. Kết quả là lịch trình tối ưu nếu tìm được.

Phương pháp thứ hai là Heuristic, kết hợp Nearest Neighbor để sinh một lời giải khởi tạo nhanh chóng, sau đó dùng 2-opt để cải tiến. Cụ thể, Nearest Neighbor sẽ chọn lần lượt khách hàng gần nhất (theo khoảng cách $d[i][j]$) mà vẫn thỏa Time Window. Sau khi đã ghé đủ n khách, 2-opt được áp dụng để hoán đổi các đoạn con trong chuỗi nhằm giảm tổng chi phí, đồng thời vẫn kiểm tra lại Time Window để đảm bảo tính khả thi. Phương pháp này nhanh chóng và phù hợp khi n lớn, nhưng không đảm bảo tối ưu tuyệt đối.

Chương trình đọc đầu vào gồm ba phần chính:

1. Dòng đầu tiên là số khách hàng n. (Depot được mặc định là 0, khách hàng từ 1 đến n.)

2. Tiếp theo là $(n+1) \times (n+1)$ số nguyên, ma trận khoảng cách $d[i][j]$ từ điểm i sang

j.

3. Sau đó có $n+1$ dòng, mỗi dòng gồm ba số thực e_i, l_i, s_i:

   - e_i, l_i là Time Window tại điểm i, nghĩa là bắt đầu phục vụ ở i chỉ được trong khoảng
   $$e_i \leq t_i \leq l_i.$$
   - s_i là thời gian phục vụ (Service Time) tại điểm i.

Sau khi đọc xong, chương trình tính ma trận thời gian di chuyển $\tau[i][j] = 1.2 \times d[i][j]$. Tiếp theo, hằng số M được khởi tạo bằng

$$M \;=\; l_0 \;+\; \max_{i,j}\{\tau_{ij}\} \;+\; \max_{i}\{s_i\},$$

với l_0 là giới hạn trên Time Window của depot. Hằng số M dùng để ràng buộc Time Window trong Branch & Bound.

Sau đó, chương trình gọi hai hàm chính:

- `solveBranchAndBound()` để tìm route tối ưu tuyệt đối bằng thuật toán Branch & Bound.
- `solveHeuristic(...)` để tìm route gần tối ưu bằng Nearest Neighbor + 2-opt.

Cuối cùng, chương trình in ra kết quả gồm:

- Phần "Branch & Bound (Exact)" hiển thị chi phí tối ưu và lịch trình tối ưu (nếu tìm được). Nếu không có lịch trình khả thi, sẽ báo "Khong tim duoc lich trinh kha thi".
- Phần "Heuristic (Nearest Neighbor + 2-opt)" hiển thị chi phí tìm được và lịch trình tương ứng (nếu đủ n khách). Nếu không ghé đủ, sẽ báo "Heuristic khong ghe du n khach" và in lịch trình đang có.

Test Case ví dụ

Dưới đây là ví dụ đầu vào "3 khách hàng" mà chương trình có thể đọc và cho kết quả như mong đợi:

```
3
0 10 12 20
10 0 8 15
12 8 0 18
20 15 18 0
```

```
0 100 0
10 50 5
0 40 5
5 60 5
```

Giải thích test case:

- Dòng `3` là $n = 3$. Depot $= 0$, khách hàng là 1, 2, 3.
- Bốn dòng tiếp theo là ma trận khoảng cách 4×4 (từ 0 đến 3):

$$\begin{pmatrix} 0 & 10 & 12 & 20 \\ 10 & 0 & 8 & 15 \\ 12 & 8 & 0 & 18 \\ 20 & 15 & 18 & 0 \end{pmatrix}.$$

- Bốn dòng sau cùng, mỗi dòng gồm $e_i\, l_i\, s_i$:
  - Depot ($i = 0$): `0 100 0` nghĩa là Time Window $[0, 100]$, service time $s_0 = 0$.
  - Khách 1: `10 50 5` nghĩa là Time Window $[10, 50]$, service time 5.
  - Khách 2: `0 40 5` nghĩa là Time Window $[0, 40]$, service time 5.
  - Khách 3: `5 60 5` nghĩa là Time Window $[5, 60]$, service time 5.

Với input này, chương trình sẽ xuất:

```
=== Branch & Bound (Exact) ===
Chi phi toi uu: 55
Lich trinh: 0 2 1 3 0

=== Heuristic (Nearest Neighbor + 2-opt) ===
Chi phi tim duoc: 55
Lich trinh: 0 2 1 3 0
```

Phần V: Lộ trình Phát triển Sản phẩm

1. **Pha 1: Phân tích & Thiết kế**
   - Xác định rõ yêu cầu: chức năng chính của hệ thống (nhập tọa độ, time windows, chọn phương pháp, hiển thị bản đồ có traffic).
   - Thiết kế kiến trúc tổng quan:
     - Frontend: Next.js + Google Maps
     - Backend: Django REST + MongoDB
     - Solver: C++ binary (Branch & Bound, Heuristic)
   - Vẽ sơ đồ luồng dữ liệu (Data Flow Diagram).
   - Thiết kế sơ lược schema MongoDB để lưu "job" (input, ma trận, kết quả).
2. **Pha 2: Xây dựng Backend Cơ bản**
   - Khởi tạo project Django và app "api".
   - Cài đặt Django REST Framework, pymongo, python-dotenv.
   - Viết kết nối MongoDB (pymongo) và tạo collection "jobs".
   - Định nghĩa serializer để validate đầu vào (`n`, `coords`, `time_windows`, `method`).
   - Xây API endpoint `POST /api/solve/`:
     - Lưu input vào MongoDB.
     - Trả về `job_id` tạm thời.
3. **Pha 3: Tích hợp Google Distance Matrix & Solver**
   - Viết hàm gọi Google Distance Matrix API, lấy `duration_in_traffic` cho mọi cặp tọa độ:
     $$\tau_{ij} = \frac{\text{duration\_in\_traffic}_{ij}}{60} \text{ (phút)}.$$
   - Viết hàm `run_solver(...)`:
     - Xuất dữ liệu vào file tạm (`n`, `distance_matrix`, `time_windows`, `method`).
     - Gọi binary C++ solver qua `subprocess`, parse stdout (`cost`, `route`).
     - Tính `arrival_times` dựa trên τ_{ij} và time windows.
   - Hoàn thiện API `/api/solve/` để:
     - Gọi Google API → xây `distance_matrix`, `travel_time_matrix`.
     - Gọi solver để thu kết quả.
     - Cập nhật document "job" với ma trận và kết quả, rồi trả JSON cuối cùng.

4. **Pha 4: Xây dựng Frontend Cơ bản**

   - Khởi tạo Next.js, cài Axios.
   - Tạo component `InputForm`:
     - Nhập `n`, `coords` (textarea), `time_windows` (textarea).
     - Chọn `method` ("exact" hoặc "heuristic").
     - Gọi callback `onSubmit(data)`.
   - Tạo trang chính `pages/index.js`:
     - Hiển thị `InputForm`.
     - Khi nhận dữ liệu, gửi POST đến `/api/solve/`.
     - Hiển thị kết quả `cost`, `route`, `arrival_times`.
   - Tích hợp Google Maps JS API vào `_app.js`.

5. **Pha 5: Vẽ Route & Traffic trên Bản đồ**

   - Tạo component `MapComponent` (chỉ render ở client):
     - Khởi tạo `google.maps.Map`.
     - Bật `TrafficLayer`.
     - Vẽ `Marker` cho mỗi điểm theo thứ tự `route`.
     - Vẽ `Polyline` nối các `coords[route[i]]`.
   - Trong `Home`, truyền `route` và `coords` vào `MapComponent`.

6. **Pha 6: Kiểm thử & Hoàn thiện**

   - Kiểm thử end-to-end local:
     - Chạy MongoDB, Django, Next.js.
     - Dùng test case ví dụ (n=3) để đảm bảo:

$$\text{cost} = 55, \quad \text{route} = [0, 2, 1, 3, 0].$$

     - Quan sát map hiển thị chính xác, traffic layer hoạt động.
   - Tinh chỉnh giao diện (CSS cơ bản, responsive).
   - Xử lý lỗi đầu vào (n không khớp số dòng coords/time windows).
   - Cải thiện UX: hiển thị spinner khi đang tính, show error message nếu API trả lỗi.

7. **Pha 7: Nâng cao & Mở rộng**

   - **Authentication / Authorization** (tùy chọn):

  - Thêm Django User, JWT (DRF Simple JWT).
  - Cho phép user xem lịch sử "jobs" đã chạy.
- **Cache kết quả Google API**:
  - Dùng Redis để lưu cặp $(\text{lat}_i, \text{lng}_i, \text{lat}_j, \text{lng}_j) \rightarrow$ giá trị `duration_in_traffic`, giảm quota call.
- **Tự động chuyển sang Heuristic khi n lớn**:
  - Nếu $n > 12$, backend tự động bỏ "exact" và dùng "heuristic" để tránh B&B quá chậm.
- **Mở rộng đa xe (VRPTW)**:
  - Thêm tham số số xe m, capacity từng xe.
  - Viết thêm heuristic phù hợp (k-means clustering & local search).
  - Frontend hiển thị các tuyến màu khác nhau cho từng xe.
- **Giám sát & Logging**:
  - Cài Sentry để theo dõi lỗi backend.
  - Lưu log chi tiết thời gian chạy solver, số API call.
  - Giám sát MongoDB size và dọn dẹp document cũ khi cần.