# Problem 1

**問題描述**

- 節點：0號 爲倉庫（depot），1–6 號爲貨物。
- 每輛車從 depot 出發(共兩輛車)，運送若干貨物後回到 depot。
- 客户有到達時間窗，車輛可以提早到並等待。
- 每個客户有服務時間（裝卸/處理時間）。

目標：最小化所有車的總行駛時間

**Assumption**

- 每輛車起訖在depot
- 可以視情況放棄某些節點
- 車輛可以在節點等待直到時間窗打開
- 行駛時間與服務時間爲定值
- 不考慮車輛容量限制

**Objective function**

$$\min \sum_{k \in V} \sum_{i \in N} \sum_{j \in N, j \neq i} c_{ij}\, x_{ij}^{k}$$

**Parameters:**

- N = {0,1,2,...,n}：節點集合，0 爲 depot
- V = {1,...,m}：車輛集合，共 m 輛車
- $c_{ij}$：節點 i 到 j 的行駛時間

**Decision Variables**

- $x_{ij}^{k} \in \{0, 1\}$：若車輛k從節點 i 開到節點 j ，爲1，否則爲0。
- $t_i$：車輛到達節點 i 的時間。

## Solution

```
from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp
```

程式碼 1: 匯入

匯入所需ortools工具

- pywrapcp：Routing 模型主要 API（建模、求解）。
- routing_enums_pb2：各種 enum（例如首解策略、局部搜尋啓發式）。

```
def create_data():
    data = {}
    # 0: depot, 1..6: customers
    data["time_matrix"] = [
        #  0  1  2  3  4  5  6
       [  0, 7, 9, 8, 6, 5,10],  # 0 depot
        [  7, 0, 4, 3, 6, 8, 7],  # 1
        [  9, 4, 0, 5, 7, 6, 4],  # 2
        [  8, 3, 5, 0, 4, 7, 6],  # 3
        [  6, 6, 7, 4, 0, 3, 5],  # 4
        [  5, 8, 6, 7, 3, 0, 4],  # 5
        [ 10, 7, 4, 6, 5, 4, 0],  # 6
    ]

# 到達時間窗 分鐘()
    data["time_windows"] = [
        (0, 10),  # 0 ：出發窗depot
        (5, 12),  # 1
        (6, 14),  # 2
        (8, 16),  # 3
        (4, 10),  # 4
        (7, 15),  # 5
        (3,  9),  # 6
    ]
# 服務時間（分鐘）
    data["service_time"] = [0, 2, 2, 2, 1, 1, 1]
    data["num_vehicles"] = 2
    data["depot"] = 0
    return data
```

程式碼 2: 參數

以上爲題目給定之參數。

```
    penalty = 1000
    for node in range(1, len(data["time_matrix"])):
        routing.AddDisjunction([manager.NodeToIndex(node)], penalty)
```

程式碼 3: 懲罰

由於原先題目給定之time matix及time windows過於嚴苛，導致求解器無法求出任一組符合條件並運送完所有地點的解，加上服務時間這個條件又將使之更難達成，因此勢必需要捨棄適當地點，所以我加上了懲罰條件(程式碼2)，讓車輛可以適時地放棄某些點位以找到可行解，但每放棄一個點位將增加成本至ArcCost中，這裡將penalty設定成極大值，讓求解器只有在解infeasible時才會選擇放棄點位。

```
# Transit （旅行時間）callback
    def time_callback(from_index, to_index):
        i = manager.IndexToNode(from_index)
        j = manager.IndexToNode(to_index)
        return data["time_matrix"][i][j] + data["service_time"][j]  # must be
    int
    transit_cb = routing.RegisterTransitCallback(time_callback)
    # 目標：路線成本用時間
    routing.SetArcCostEvaluatorOfAllVehicles(transit_cb)
```

程式碼 4: transit callback

- Transit Callback：告訴求解器每條路線的成本（此處就是旅行時間）。
- SetArcCostEvaluatorOfAllVehicles：把這個成本函式當作目標(所有車旅行時間和+懲罰)。

這裡假設每個點位的開放時間窗僅限制車輛的到達時間，所以將每個終點的服務時間加入 time callback 中，所以 time callback 就會變成 i 到 j 的所需時間加上 j 點的服務時間。

```
    # 時間維度（允許等待、放寬容量）
    time_name = "Time"
    LARGE = 10**6
    routing.AddDimension(
        transit_cb,
        LARGE,  # waiting slack
        LARGE,  # capacity (per-vehicle horizon)
        False,  # don't force start at t=0
        time_name,
    )
    time_dim = routing.GetDimensionOrDie(time_name)
```

程式碼 5: 時間維度設定

- AddDimension 會爲每個「索引」建立一個累積變數 CumulVar（這裡代表到達時間）。
- slack_max 大代表可等待（早到可以等到時間窗打開）。
- capacity 大代表行程總時間不限。
- False：起點時間不強制 0；用 depot 視窗去限定起點。

```python
    # 非 depot 節點：套時間窗
    for node, (open_t, close_t) in enumerate(data["time_windows"]):
        if node == data["depot"]:
            continue
        index = manager.NodeToIndex(node)
        time_dim.CumulVar(index).SetRange(open_t, close_t)
```

程式碼 6: 套非depot時間窗

- 每個（非 depot）節點都有一個 CumulVar，要求其值 [open_t, close_t]。
- 因爲允許等待，車可以提前到達，CumulVar 會被推到 open_t。

```python
    # 起點：套 depot 出發窗
    depot_open, depot_close = data["time_windows"][data["depot"]]
    for v in range(data["num_vehicles"]):
        start_idx = routing.Start(v)
        time_dim.CumulVar(start_idx).SetRange(depot_open, depot_close)
```

程式碼 7: 套 depot 出發窗

- 每輛車的Start 索引也有一個 CumulVar，把它限制在 depot 視窗。

```python
    # 終點：放寬（重要）
    for v in range(data["num_vehicles"]):
        end_idx = routing.End(v)
        time_dim.CumulVar(end_idx).SetRange(0, LARGE)
```

程式碼 8: 套終點時間窗

- 這裡把 End 放寬，表示「回到 depot 不受時間窗限制」。

```python
    # ：讓Finalizers start/end 偏好較小
    for v in range(data["num_vehicles"]):
        routing.AddVariableMinimizedByFinalizer(time_dim.CumulVar(routing.Start
    (v)))
        routing.AddVariableMinimizedByFinalizer(time_dim.CumulVar(routing.End(v
    )))
```

程式碼 9: finalizer

- Finalizer 在找到可行解後，會把這些變數往小的方向微調（有助產生漂亮/緊湊的時間表）。

```python
    # 搜尋參數
    search_params = pywrapcp.DefaultRoutingSearchParameters()
    search_params.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC
    )
    search_params.local_search_metaheuristic = (
        routing_enums_pb2.LocalSearchMetaheuristic.GUIDED_LOCAL_SEARCH
    )
    search_params.time_limit.FromSeconds(10)

    # 求解
    solution = routing.SolveWithParameters(search_params)
    if solution:
        print_solution(data, manager, routing, solution)
    else:
        print("No solution found.")
```

程式碼 10: 求解

- PATH_CHEAPEST_ARC：先用「接最便宜的弧」貪婪法找一組首解。
- GUIDED_LOCAL_SEARCH：在首解附近做結構性調整（翻轉、交換等）以跳脫區域極小。
- time_limit：限制求解時間（秒），保證可回傳目前最佳解（若有）。

```python
def get_cumul_data(solution, routing, dimension):
    """Return per-route cumulative min/max arrays for a given dimension."""
    cumul_data = []
    for v in range(routing.vehicles()):
        route_data = []
        index = routing.Start(v)
        dim_var = dimension.CumulVar(index)
        route_data.append([solution.Min(dim_var), solution.Max(dim_var)])
        while not routing.IsEnd(index):
            index = solution.Value(routing.NextVar(index))
            dim_var = dimension.CumulVar(index)
            route_data.append([solution.Min(dim_var), solution.Max(dim_var)])
        cumul_data.append(route_data)
    return cumul_data
```

程式碼 11: 累積數據

- solution.Min(var) / solution.Max(var) 是該變數在解中的合法範圍。

```python
def print_solution(data, manager, routing, solution):
    """Prints solution on console."""
    print(f"Objective: {solution.ObjectiveValue()}")
    time_dimension = routing.GetDimensionOrDie("Time")

    # 也可取得 Min/Max 範圍版本
    cumul_bounds = get_cumul_data(solution, routing, time_dimension)

    total_time_sum = 0
    makespan = 0

    for vehicle_id in range(data["num_vehicles"]):
        index = routing.Start(vehicle_id)
        plan = []
        while not routing.IsEnd(index):
            node = manager.IndexToNode(index)
            arrive = solution.Value(time_dimension.CumulVar(index))
            plan.append(f"{node} (arrive {arrive}, window={data['time_windows
    ']][node]})")
            index = solution.Value(routing.NextVar(index))
        # （回到End ），一般不限制視窗depot
        node = manager.IndexToNode(index)
        arrive = solution.Value(time_dimension.CumulVar(index))
        plan.append(f"{node} (arrive {arrive})")

        print(f"Route for vehicle {vehicle_id}:\n  " + " -> ".join(plan))
        print(f"  Route time: {arrive}min\n")

        total_time_sum += arrive
        makespan = max(makespan, arrive)

    print(f"Total time (sum of route end times): {total_time_sum}min")
    print(f"Makespan   (max route end time):    {makespan}min")

    # 若想看 Min/Max 範圍，可解除註解：
    # for i, route_bounds in enumerate(cumul_bounds):
    #     print(f"[Bounds] Vehicle {i}: " + " -> ".join(f"({a},{b})" for a,b in
    route_bounds))
if __name__ == "__main__":
    main()
```

程式碼 12: 輸出解

- IndexToNode(index)：把內部索引還原爲外部節點編號（便於看懂路線）。
- NextVar(index)：路徑上的下一個索引。
- CumulVar(index)：此索引的「時間維度」累積值（到達時間）。
- Route time 用終點（End）的 CumulVar 值（單位與 time_matrix 相同）。
- Total time 是所有車路徑終點時間的加總；Makespan 是它們之中最大值。

# Result

```
Objective: 2038
Dropped nodes: 2 6
Route for vehicle 0:
  0 (arrive 0, window=(0, 10)) -> 4 (arrive 7, window=(4, 10)) -> 5 (arrive 11, window=(7, 15)) -> 0 (arrive 16)
  Route time: 16min

Route for vehicle 1:
  0 (arrive 0, window=(0, 10)) -> 1 (arrive 9, window=(5, 12)) -> 3 (arrive 14, window=(8, 16)) -> 0 (arrive 22)
  Route time: 22min

Total time (sum of route end times): 38min
Makespan   (max route end time):    22min
```

圖 1: 執行結果

最終執行結果如圖1所示，可以看到最終求解器放棄了兩個節點 (2,6)，而懲罰值則加在 objective 中，由於懲罰設的極大，可知此兩點爲必要的放棄，最終兩輛車的總花費時間加總爲 38 min，最長路徑時間花費 22 min。

# Conclusion

本次作業透過 OR-Tools 建立車輛途程問題（VRPTW）模型，成功實現多車輛、時間窗及服務時間之整合求解，由於原始問題過於嚴苛，加入懲罰式放棄節點機制後，求解器得以在有限時間內找到可行解，並確保僅在必要時放棄節點，以維持總成本最小化，最終結果顯示，兩輛車能有效分配路徑，完成大部分貨物的配送，總行駛時間爲 38 分鐘，且最大路徑時間爲 22 分鐘，達成了最小化總行駛時間的目標。

# References

1. Google, “OR-Tools: Operations Research Tools.” Google Developers. [Online]. Available: https://developers.google.com/optimization. [Accessed: Sep. 14, 2025].