```python
import gurobipy as gp
from gurobipy import GRB
import numpy as np
import time
from collections import defaultdict
import pandas as pd

class LAArcComputer:
    def __init__(self, time_windows, travel_times):
        """
        Initialize LA Arc Computer

        Args:
            time_windows: Dict[int, Tuple[float, float]] - For each customer, (earliest, latest) service times
            travel_times: Dict[Tuple[int, int], float] - Travel time between each pair of customers
        """
        self.time_windows = time_windows
        self.travel_times = travel_times

    def compute_phi_r(self, r: list) -> float:
        """
        Compute φ_r: earliest time we could leave first customer without waiting
        """
        print(f"\nComputing φ_r for sequence {r}")
        if len(r) == 2:  # Base case (13a)
            u, v = r
            t_minus_u = self.time_windows[u][0]
            t_minus_v = self.time_windows[v][0]
            t_uv = self.travel_times[u,v]
            result = max(t_minus_u, t_minus_v - t_uv)
            print(f"Base case φ_r:")
            print(f"  Customer {u} earliest time (t_minus_u): {t_minus_u}")
            print(f"  Customer {v} earliest time (t_minus_v): {t_minus_v}")
            print(f"  Travel time (t_uv): {t_uv}")
            print(f"  Result: max({t_minus_u}, {t_minus_v} - {t_uv}) = {result}")
            return result

        else:  # Recursive case (13c)
            u = r[0]
            w = r[1]
            t_minus_u = self.time_windows[u][0]
            t_uw = self.travel_times[u,w]
            print(f"Recursive case for {r}:")
            print(f"  Computing φ_r for subsequence {r[1:]}")
            phi_r_minus = self.compute_phi_r(r[1:])
```

```python
            result = max(t_minus_u, phi_r_minus - t_uw)  # Propagate time backwards
            print(f"  Customer {u} earliest time (t_minus_u): {t_minus_u}")
            print(f"  Travel time to {w} (t_uw): {t_uw}")
            print(f"  Recursive result (phi_r_minus): {phi_r_minus}")
            print(f"  Final result: max({t_minus_u}, {phi_r_minus} - {t_uw}) = {result}")
            return result

    def compute_phi_hat_r(self, r: list) -> float:
        """
        Compute φ̂_r: latest feasible departure time from first customer
        """
        print(f"\nComputing φ̂_r for sequence {r}")
        if len(r) == 2:  # Base case (13b)
            u, v = r
            t_plus_u = self.time_windows[u][1]
            t_minus_v = self.time_windows[v][0]  # Need earliest time at v
            t_uv = self.travel_times[u,v]
            result = min(t_plus_u, t_minus_v - t_uv)
            print(f"Base case φ̂_r:")
            print(f"  Customer {u} latest time (t_plus_u): {t_plus_u}")
            print(f"  Customer {v} earliest time (t_minus_v): {t_minus_v}")
            print(f"  Travel time (t_uv): {t_uv}")
            print(f"  Result: min({t_plus_u}, {t_minus_v} - {t_uv}) = {result}")
            return result

        else:  # Recursive case (13d)
            u = r[0]
            w = r[1]
            t_plus_u = self.time_windows[u][1]
            t_uw = self.travel_times[u,w]
            print(f"Recursive case for {r}:")
            print(f"  Computing φ̂_r for subsequence {r[1:]}")
            phi_hat_r_minus = self.compute_phi_hat_r(r[1:])
            result = min(t_plus_u, phi_hat_r_minus - t_uw)  # Propagate time backwards
            print(f"  Customer {u} latest time (t_plus_u): {t_plus_u}")
            print(f"  Travel time to {w} (t_uw): {t_uw}")
            print(f"  Recursive result (phi_hat_r_minus): {phi_hat_r_minus}")
            print(f"  Final result: min({t_plus_u}, {phi_hat_r_minus} - {t_uw}) = {result}")
            return result

class VRPTWOptimizer:
    def __init__(self, customers, depot_start, depot_end, costs, time_windows, service_times, demands,
```

```
                 vehicle_capacity, K=3, time_granularity=3, capacity_granularity=3, max_iterations=5):
        self.customers = customers
        self.depot_start = depot_start
        self.depot_end = depot_end
        self.costs = costs
        self.time_windows = time_windows
        self.service_times = service_times  # Added service times
        self.demands = demands
        self.vehicle_capacity = vehicle_capacity
        self.K = K
        self.time_granularity = time_granularity
        self.capacity_granularity = capacity_granularity
        self.max_iterations = max_iterations

        # Algorithm parameters remain the same
        self.MIN_INC = 1
        self.sigma = 9

        self.model = None
        self.create_initial_model()

    def _add_constraints(self):
        """Add all constraints"""
        # Objective function remains unchanged - uses costs (distances) only
        self.model.setObjective(
            gp.quicksum(self.costs[i,j] * self.x[i,j] for i,j in self.E_star),
            GRB.MINIMIZE
        )

        # Visit each customer once constraints remain unchanged
        for u in self.customers:
            self.model.addConstr(
                gp.quicksum(self.x[i,u] for i,j in self.E_star if j == u) == 1,
                name=f'visit_in_{u}'
            )
            self.model.addConstr(
                gp.quicksum(self.x[u,j] for i,j in self.E_star if i == u) == 1,
                name=f'visit_out_{u}'
            )

        # Time window constraints updated to include service times
        M = max(tw[1] for tw in self.time_windows.values())
        for (i,j) in self.E_star:
            if j != self.depot_end:
```

```
                self.model.addConstr(
                    self.tau[j] >= self.tau[i] + self.service_times[i] +
self.costs[i,j]/5
                    - M * (1 - self.x[i,j]),
                    name=f'time_prop_{i}_{j}'
                )

        # Time window bounds remain unchanged
        for i in self.customers + [self.depot_start, self.depot_end]:
            self.model.addConstr(
                self.tau[i] >= self.time_windows[i][0],
                name=f'tw_lb_{i}'
            )
            self.model.addConstr(
                self.tau[i] <= self.time_windows[i][1],
                name=f'tw_ub_{i}'
            )

        # Other constraints remain unchanged
        self._add_la_arc_constraints_with_parsimony()
        self._add_capacity_constraints()
        self._add_time_flow_constraints()
        self._add_capacity_flow_constraints()
        self.model.update()

    def create_initial_model(self):
        """Create initial model with all components"""
        self.model = gp.Model("VRPTW")

        # Create valid edges
        self.E_star = [(i,j) for i in [self.depot_start] + self.customers
                    for j in self.customers + [self.depot_end] if i != j]

        # Generate LA neighborhoods
        self.la_neighbors = self._generate_initial_la_neighbors()

        # Generate orderings
        self.R_u = self._generate_orderings()

        # Create time and capacity discretization
        self.T_u = self._create_time_buckets()
        self.D_u = self._create_capacity_buckets()

        # Create flow graphs first
        self.nodes_T, self.edges_T = self._create_time_graph()
        self.nodes_D, self.edges_D = self._create_capacity_graph()

        # Then create variables that depend on the graphs
        self._create_variables()
```

```python

        # Finally add constraints
        self._add_constraints()

    def _create_variables(self):
        """Create all decision variables"""
        # Route variables x_{ij}
        self.x = {}
        for i, j in self.E_star:
            self.x[i,j] = self.model.addVar(vtype=GRB.BINARY, name=f'x_{i}_{j}')

        # Time variables τ_i
        self.tau = {}
        for i in self.customers + [self.depot_start, self.depot_end]:
            self.tau[i] = self.model.addVar(lb=0, name=f'tau_{i}')

        # LA-arc variables y_r
        self.y = {}
        for u in self.customers:
            for r in range(len(self.R_u[u])):
                self.y[u,r] = self.model.addVar(vtype=GRB.BINARY, name=f'y_{u}_{r}')

        # Time flow variables z_T
        self.z_T = {}
        for edge in self.edges_T:
            self.z_T[edge] = self.model.addVar(lb=0, name=f'z_T_{edge}')

        # Capacity flow variables z_D
        self.z_D = {}
        for edge in self.edges_D:
            self.z_D[edge] = self.model.addVar(lb=0, name=f'z_D_{edge}')

        # Load tracking variables
        self.load = {}
        for i in self.customers:
            self.load[i] = self.model.addVar(lb=0, ub=self.vehicle_capacity, name=f'load_{i}')

        self.model.update()

    def _add_la_arc_constraints_with_parsimony(self):
        """Add LA-arc movement consistency constraints with parsimony penalties"""
        # Small positive value for parsimony penalties
        rho = 0.01  # ϱ in the paper

        # For each customer u, track customers by distance
```

```
        for u in self.customers:
            # Get distances to all other customers
            distances = [(j, self.costs[u,j]) for j in self.customers if j != u]
            distances.sort(key=lambda x: x[1])

            # Create k-indexed neighborhoods (N^k_u)
            N_k_u = {}  # k-indexed neighborhoods
            N_k_plus_u = {}  # N^k_+u from paper (includes u)
            for k in range(1, len(distances) + 1):
                N_k_u[k] = [j for j, _ in distances[:k]]
                N_k_plus_u[k] = [u] + N_k_u[k]

            # Select one ordering per customer (equation 6g)
            self.model.addConstr(
                gp.quicksum(self.y[u,r] for r in range(len(self.R_u[u]))) == 1,
                name=f'one_ordering_{u}'
            )

            # Add k-indexed constraints from equation (8a)
            for k in range(1, len(distances) + 1):
                for w in N_k_plus_u[k]:
                    for v in N_k_u[k]:
                        if (w,v) in self.E_star:
                            self.model.addConstr(
                                rho * k + self.x[w,v] >= gp.quicksum(
                                    self.y[u,r] for r in range(len(self.R_u[u]))
                                    if self._is_in_k_neighborhood_ordering(r, w, v, k, N_k_u[k])
                                ),
                                name=f'la_arc_cons_{u}_{w}_{v}_{k}'
                            )

            # Add k-indexed constraints from equation (8b)
            for k in range(1, len(distances) + 1):
                for w in N_k_plus_u[k]:
                    outside_neighbors = [j for j in self.customers + [self.depot_end]
                                      if j not in N_k_plus_u[k]]
                    if outside_neighbors:
                        self.model.addConstr(
                            rho * k + gp.quicksum(
                                self.x[w,j] for j in outside_neighbors
                                if (w,j) in self.E_star
                            ) >= gp.quicksum(
                                self.y[u,r] for r in range(len(self.R_u[u]))
```

```
                                if self._is_final_in_k_neighborhood(r, w, k, N_k_plus_u[k])
                            ),
                            name=f'la_arc_final_{u}_{w}_{k}'
                        )

    def _add_capacity_constraints(self):
        """Add capacity flow constraints"""
        # Initial load from depot
        for j in self.customers:
            self.model.addConstr(
                self.load[j] >= self.demands[j] * self.x[self.depot_start,j],
                name=f'init_load_{j}'
            )

        # Load propagation between customers
        M = self.vehicle_capacity
        for i in self.customers:
            for j in self.customers:
                if i != j and (i,j) in self.E_star:
                    self.model.addConstr(
                        self.load[j] >= self.load[i] + self.demands[j] - M * (1 - self.x[i,j]),
                        name=f'load_prop_{i}_{j}'
                    )

        # Enforce capacity limit
        for i in self.customers:
            self.model.addConstr(
                self.load[i] <= self.vehicle_capacity,
                name=f'cap_limit_{i}'
            )

    def _add_capacity_flow_constraints(self):
        """Add capacity flow constraints (equations 4a and 4b from paper)"""
        # Flow conservation (4a)
        for i, k, d_min, d_max in self.nodes_D:
            if i not in [self.depot_start, self.depot_end]:
                self.model.addConstr(
                    gp.quicksum(self.z_D[e] for e in self.edges_D if e[0] == (i,k)) ==
                    gp.quicksum(self.z_D[e] for e in self.edges_D if e[1] == (i,k)),
                    name=f'cap_flow_cons_{i}_{k}'
                )

        # Consistency with route variables (4b)
```

```python
        for u,v in self.E_star:
            self.model.addConstr(
                self.x[u,v] == gp.quicksum(
                    self.z_D[e] for e in self.edges_D
                    if e[0][0] == u and e[1][0] == v
                ),
                name=f'cap_flow_cons_route_{u}_{v}'
            )

    def _add_time_flow_constraints(self):
        """Add time flow constraints"""
        # Flow conservation (5a)
        for i, k, t_min, t_max in self.nodes_T:
            if i not in [self.depot_start, self.depot_end]:
                self.model.addConstr(
                    gp.quicksum(self.z_T[e] for e in self.edges_T if e[0] == (i,k)) ==
                    gp.quicksum(self.z_T[e] for e in self.edges_T if e[1] == (i,k)),
                    name=f'time_flow_cons_{i}_{k}'
                )

        # Consistency with route variables (5b)
        for u,v in self.E_star:
            self.model.addConstr(
                self.x[u,v] == gp.quicksum(
                    self.z_T[e] for e in self.edges_T
                    if e[0][0] == u and e[1][0] == v
                ),
                name=f'time_flow_cons_route_{u}_{v}'
            )

        # Link time variables τ with time buckets
        M = max(tw[1] for tw in self.time_windows.values())
        for i, k, t_min, t_max in self.nodes_T:
            if i not in [self.depot_start, self.depot_end]:
                outgoing_edges = [e for e in self.edges_T if e[0] == (i,k)]
                if outgoing_edges:
                    self.model.addConstr(
                        self.tau[i] >= t_min - M * (1 - gp.quicksum(self.z_T[e] for e in outgoing_edges)),
                        name=f'time_bucket_lb_{i}_{k}'
                    )
                    self.model.addConstr(
                        self.tau[i] <= t_max + M * (1 - gp.quicksum(self.z_T[e] for e in outgoing_edges)),
                        name=f'time_bucket_ub_{i}_{k}'
                    )

```

```
    def solve_with_parsimony(self, time_limit=None):
        """Solve VRPTW with LA neighborhood parsimony"""
        print("\n=== Initial LA-Neighborhood Analysis ===")
        self._print_neighborhood_analysis()

        if time_limit:
            self.model.setParam('TimeLimit', time_limit)

        # Set other Gurobi parameters
        self.model.setParam('MIPGap', 0.01)  # 1% optimality gap
        self.model.setParam('Threads', 4)     # Use 4 threads

        iteration = 1
        last_lp_val = float('-inf')
        iter_since_reset = 0

        while iteration <= self.max_iterations:
            print(f"\n=== Iteration {iteration} ===")

            # Solve current iteration
            self.model.optimize()

            current_obj = self.model.objVal if self.model.Status in [GRB.OPTIMAL, GRB.TIME_LIMIT] else None

            print(f"\nIteration Objective: {current_obj}")

            if current_obj is not None and current_obj > last_lp_val + self.MIN_INC:
                print("Solution improved")
                last_lp_val = current_obj
                iter_since_reset = 0

                try:
                    # Get dual variables and analyze buckets
                    print("\n--- Getting LP Relaxation Information ---")
                    dual_vars = self.get_dual_variables()

                    if dual_vars:  # Only analyze if we got dual variables
                        print("\n--- LA-Neighborhood Analysis After Improvement ---")
                        self._print_neighborhood_analysis()
                        print("\n--- Bucket Analysis ---")
                        self._print_bucket_analysis()
                except Exception as e:
                    print(f"Warning: Could not complete analysis due to: {e}")

            else:
```

```python
                print("No significant improvement")
                iter_since_reset += 1

            if iter_since_reset >= self.sigma:
                print("\nResetting neighborhoods to maximum size...")
                self._reset_neighborhoods()
                iter_since_reset = 0

            iteration += 1

        # Extract final solution
        solution = self._extract_solution()
        if solution['status'] == 'Optimal':
            self.validate_solution(solution)

        return solution

    def _print_neighborhood_analysis(self):
        """Print detailed analysis of LA neighborhoods"""
        print("\nLA-Neighborhood Details:")

        # Print sizes and members
        for u in self.customers:
            neighbors = self.la_neighbors[u]
            print(f"\nCustomer {u}:")
            print(f"  Neighborhood size: {len(neighbors)}")
            print(f"  Neighbors: {neighbors}")

            # Print distances to neighbors
            distances = [(j, self.costs[u,j]) for j in neighbors]
            distances.sort(key=lambda x: x[1])
            print("  Distances to neighbors:")
            for j, dist in distances:
                print(f"    -> Customer {j}: {dist/5:.1f}")

            # Print time window compatibility
            print("  Time window compatibility:")
            for j in neighbors:
                u_early, u_late = self.time_windows[u]
                j_early, j_late = self.time_windows[j]
                travel_time = self.costs[u,j]/5
                print(f"    -> Customer {j}: Window [{j_early}, {j_late}]")
                print(f"       Earliest possible arrival: {u_early + travel_time:.1f}")
                print(f"       Latest possible arrival: {u_late + travel_time:.1f}")

    def _print_bucket_analysis(self):
        """Print detailed analysis of time and capacity buckets"""
```

```
        print("\nTime Bucket Analysis:")
        for u in self.customers:
            print(f"\nCustomer {u}:")
            print(f"  Time window: {self.time_windows[u]}")
            print(f"  Number of buckets: {len(self.T_u[u])}")
            print("  Bucket ranges:")
            for i, (t_min, t_max) in enumerate(self.T_u[u]):
                print(f"    Bucket {i}: [{t_min:.1f}, {t_max:.1f}]")

        print("\nCapacity Bucket Analysis:")
        for u in self.customers:
            print(f"\nCustomer {u}:")
            print(f"  Demand: {self.demands[u]}")
            print(f"  Number of buckets: {len(self.D_u[u])}")
            print("  Bucket ranges:")
            for i, (d_min, d_max) in enumerate(self.D_u[u]):
                print(f"    Bucket {i}: [{d_min:.1f}, {d_max:.1f}]")

    def _extract_solution(self):
        """Extract solution details"""
        if self.model.Status == GRB.OPTIMAL:
            status = 'Optimal'
        elif self.model.Status == GRB.TIME_LIMIT:
            status = 'TimeLimit'
        else:
            status = 'Other'

        solution = {
            'status': status,
            'objective': self.model.ObjVal if status in ['Optimal', 'TimeLimit'] else None,
            'routes': self._extract_routes() if status in ['Optimal', 'TimeLimit'] else None,
            'computation_time': self.model.Runtime,
            'neighborhood_sizes': {u: len(neighbors) for u, neighbors in self.la_neighbors.items()}
        }

        return solution

    def _extract_routes(self):
            """Extract routes from solution"""
            if self.model.Status not in [GRB.OPTIMAL, GRB.TIME_LIMIT]:
                return None

            active_edges = [(i,j) for (i,j) in self.E_star
                        if self.x[i,j].X > 0.5]

            routes = []
```

```python
            depot_starts = [(i,j) for (i,j) in active_edges if i == self.depot_start]

            for start_edge in depot_starts:
                route = []
                current = start_edge[1]
                route.append(current)

                while current != self.depot_end:
                    next_edges = [(i,j) for (i,j) in active_edges if i == current]
                    if not next_edges:
                        break
                    current = next_edges[0][1]
                    if current != self.depot_end:
                        route.append(current)

                routes.append(route)

            return routes

    def _generate_initial_la_neighbors(self):
        """Generate initial LA neighborhoods using K closest neighbors"""
        la_neighbors = {}
        for u in self.customers:
            # Get distances to all other customers
            distances = [(j, self.costs[u,j]) for j in self.customers if j != u]
            distances.sort(key=lambda x: x[1])

            # Take K closest neighbors that are reachable
            neighbors = []
            for j, _ in distances:
                if len(neighbors) >= self.K:
                    break
                if self._is_reachable(u, j):
                    neighbors.append(j)
            la_neighbors[u] = neighbors
        return la_neighbors

    def _is_reachable(self, i, j):
        """Check if j is reachable from i considering time windows and capacity"""
        earliest_i, latest_i = self.time_windows[i]
        earliest_j, latest_j = self.time_windows[j]
        travel_time = self.costs[i,j] / 5

        if earliest_i + travel_time > latest_j:
            return False
```

```python

        if self.demands[i] + self.demands[j] > self.vehicle_capacity:
            return False

        return True

    def _generate_orderings(self):
        """Generate efficient orderings for each customer"""
        R_u = defaultdict(list)

        for u in self.customers:
            # Base ordering
            R_u[u].append({
                'sequence': [u],
                'a_wv': {},
                'a_star': {u: 1}
            })

            # Add single neighbor orderings
            for v in self.la_neighbors[u]:
                if self._is_reachable(u, v):
                    R_u[u].append({
                        'sequence': [u, v],
                        'a_wv': {(u,v): 1},
                        'a_star': {v: 1}
                    })

            # Add two-neighbor orderings
            for v1 in self.la_neighbors[u]:
                for v2 in self.la_neighbors[u]:
                    if v1 != v2 and self._is_sequence_feasible([u, v1, v2]):
                        R_u[u].append({
                            'sequence': [u, v1, v2],
                            'a_wv': {(u,v1): 1, (v1,v2): 1},
                            'a_star': {v2: 1}
                        })

        return R_u

    def _is_sequence_feasible(self, sequence):
        """Check if a sequence of customers is feasible, updated to include service times"""
        # Check capacity constraints
        total_demand = sum(self.demands[i] for i in sequence)
        if total_demand > self.vehicle_capacity:
            return False

        # Check time feasibility including service times
```

```python
        current_time = self.time_windows[sequence[0]][0]  # Start at earliest possible time

        for i in range(len(sequence)):
            current = sequence[i]

            # Cannot arrive after latest time window
            if current_time > self.time_windows[current][1]:
                return False

            # Update time to include service
            current_time = max(current_time, self.time_windows[current][0])
            current_time += self.service_times[current]

            # Add travel time to next customer if any
            if i < len(sequence)-1:
                next_customer = sequence[i+1]
                current_time += self.costs[current,next_customer]/5

        return True

    def _is_in_k_neighborhood_ordering(self, r, w, v, k, N_k_u):
        """Check if w immediately precedes v in ordering r within k-neighborhood"""
        for u in self.R_u:
            if r < len(self.R_u[u]):
                ordering = self.R_u[u][r]
                sequence = ordering['sequence']

                # Both customers must be in k-neighborhood
                if w not in N_k_u or v not in N_k_u:
                    return False

                # Check if w immediately precedes v
                for i in range(len(sequence)-1):
                    if sequence[i] == w and sequence[i+1] == v:
                        return True
                return False
        return False

    def _is_final_in_k_neighborhood(self, r, w, k, N_k_plus_u):
        """Check if w is final customer in k-neighborhood for ordering r"""
        for u in self.R_u:
            if r < len(self.R_u[u]):
                ordering = self.R_u[u][r]
                sequence = ordering['sequence']

                # w must be in k-neighborhood
                if w not in N_k_plus_u:
```

```
                    return False

                # Find position of w in sequence
                try:
                    w_pos = sequence.index(w)
                except ValueError:
                    return False

                # Check if w is last in sequence or followed by customer outside k-neighborhood
                return (w_pos == len(sequence)-1 or
                        sequence[w_pos+1] not in N_k_plus_u)
        return False

    def validate_solution(self, solution):
        """Validate solution feasibility, updated to include service times"""
        if solution['status'] != 'Optimal':
            return False

        routes = solution['routes']
        print("\nValidating solution:")

        for idx, route in enumerate(routes, 1):
            print(f"\nRoute {idx}: {' -> '.join(map(str, [self.depot_start] + route + [self.depot_end]))}")

            # Check capacity
            route_load = sum(self.demands[i] for i in route)
            print(f"  Load: {route_load}/{self.vehicle_capacity}", end=" ")
            if route_load > self.vehicle_capacity:
                print("❌ Exceeds capacity!")
                return False
            print("✓")

            # Check time windows including service times
            current_time = 0
            current_loc = self.depot_start
            for stop in route:
                travel_time = self.costs[current_loc, stop] / 5
                arrival_time = current_time + travel_time
                service_start = max(arrival_time, self.time_windows[stop][0])

                window_start, window_end = self.time_windows[stop]
                print(f"  Customer {stop}:")
                print(f"    Arrive: {arrival_time:.1f}")
                print(f"    Service start: {service_start:.1f}")
                print(f"    Window: [{window_start}, {window_end}]", end="
```

```
")

                if service_start > window_end:
                    print("❌ Misses window!")
                    return False
                print("✓")

                # Update current time to include service
                current_time = service_start + self.service_times[stop]
                current_loc = stop

        return True

    def get_dual_variables(self):
        """Get dual variables from the LP relaxation"""
        # For Gurobi, we need to solve the LP relaxation first
        relaxed = self.model.copy()
        for v in relaxed.getVars():
            if v.vType != GRB.CONTINUOUS:
                v.vType = GRB.CONTINUOUS

        relaxed.optimize()

        # Get dual values using Pi attribute
        dual_vars = {}
        if relaxed.Status == GRB.OPTIMAL:
            for c in relaxed.getConstrs():
                try:
                    dual_vars[c.ConstrName] =
relaxed.getConstrByName(c.ConstrName).Pi
                except Exception:
                    # If we can't get the dual for a constraint, skip it
                    continue

        return dual_vars

    def _create_capacity_buckets(self):
        """Create initial capacity buckets for each customer"""
        buckets = {}

        # Create buckets for each customer
        for u in self.customers:
            demand = self.demands[u]
            remaining_capacity = self.vehicle_capacity - demand
            bucket_size = remaining_capacity / self.capacity_granularity

            customer_buckets = []
            current_capacity = demand

```

```python
            # Create evenly spaced buckets
            for i in range(self.capacity_granularity):
                lower = current_capacity
                upper = min(current_capacity + bucket_size, self.vehicle_capacity)
                customer_buckets.append((lower, upper))
                current_capacity = upper

                if current_capacity >= self.vehicle_capacity:
                    break

            buckets[u] = customer_buckets

        # Add single bucket for depot (start and end)
        buckets[self.depot_start] = [(0, self.vehicle_capacity)]
        buckets[self.depot_end] = [(0, self.vehicle_capacity)]

        return buckets

    def _create_capacity_graph(self):
        """Create directed graph GD for capacity flow"""
        nodes_D = []  # (u, k, d⁻, d⁺)
        edges_D = []  # ((u1,k1), (u2,k2))

        # Create nodes for each customer and their capacity buckets
        for u in self.customers:
            for k, (d_min, d_max) in enumerate(self.D_u[u]):
                nodes_D.append((u, k, d_min, d_max))

        # Add depot nodes
        depot_start_bucket = self.D_u[self.depot_start][0]
        depot_end_bucket = self.D_u[self.depot_end][0]
        nodes_D.append((self.depot_start, 0, depot_start_bucket[0], depot_start_bucket[1]))
        nodes_D.append((self.depot_end, 0, depot_end_bucket[0], depot_end_bucket[1]))

        # Create edges between nodes
        for i, k_i, d_min_i, d_max_i in nodes_D:
            for j, k_j, d_min_j, d_max_j in nodes_D:
                if i != j:
                    # Check if edge is feasible based on capacity
                    demand_j = self.demands[j]
                    remaining_i = d_max_i - self.demands[i]

                    # Edge is feasible if remaining capacity after i can accommodate j
                    if (remaining_i >= demand_j and
                        d_min_j >= demand_j and
```

```
                        d_max_j <= self.vehicle_capacity):
                        edges_D.append(((i,k_i), (j,k_j)))

        return nodes_D, edges_D

    def _create_time_buckets(self):
        """Create initial time buckets for each customer"""
        buckets = {}

        # Create buckets for each customer
        for u in self.customers:
            earliest_time, latest_time = self.time_windows[u]
            time_span = latest_time - earliest_time
            bucket_size = time_span / self.time_granularity

            customer_buckets = []
            current_time = earliest_time

            # Create evenly spaced buckets
            for i in range(self.time_granularity):
                lower = current_time
                upper = min(current_time + bucket_size, latest_time)
                customer_buckets.append((lower, upper))
                current_time = upper

                if current_time >= latest_time:
                    break

            buckets[u] = customer_buckets

        # Add single bucket for depot (start and end)
        depot_earliest, depot_latest = self.time_windows[self.depot_start]
        buckets[self.depot_start] = [(depot_earliest, depot_latest)]
        buckets[self.depot_end] = [(depot_earliest, depot_latest)]

        return buckets

    def _create_time_graph(self):
        """Create directed graph GT for time flow, updated to include service times"""
        nodes_T = []  # (u, k, t⁻, t⁺)
        edges_T = []  # ((u1,k1), (u2,k2))

        # Create nodes for each customer and their time buckets
        for u in self.customers:
            for k, (t_min, t_max) in enumerate(self.T_u[u]):
                nodes_T.append((u, k, t_min, t_max))

        # Add depot nodes
```

```
        depot_start_bucket = self.T_u[self.depot_start][0]
        depot_end_bucket = self.T_u[self.depot_end][0]
        nodes_T.append((self.depot_start, 0, depot_start_bucket[0], depot_start_bucket[1]))
        nodes_T.append((self.depot_end, 0, depot_end_bucket[0], depot_end_bucket[1]))

        # Create edges between nodes, including service times in calculations
        for i, k_i, t_min_i, t_max_i in nodes_T:
            for j, k_j, t_min_j, t_max_j in nodes_T:
                if i != j:
                    travel_time = self.costs[i,j] / 5
                    service_time = self.service_times[i]  # Service time at origin

                    earliest_arrival = t_min_i + service_time + travel_time
                    latest_arrival = t_max_i + service_time + travel_time

                    if (earliest_arrival <= t_max_j and
                        latest_arrival >= t_min_j and
                        earliest_arrival <= self.time_windows[j][1] and
                        latest_arrival >= self.time_windows[j][0]):
                        edges_T.append(((i,k_i), (j,k_j)))

        return nodes_T, edges_T

    def _merge_capacity_buckets(self, dual_vars):
        """Merge capacity buckets when their dual variables are equal"""
        print("\nMerging capacity buckets...")
        for u in self.customers:
            print(f"\nCustomer {u}:")
            print(f"  Before merge: {self.D_u[u]}")

            buckets_to_merge = []

            # Get all consecutive bucket pairs for this customer
            for k in range(len(self.D_u[u]) - 1):
                i = (u, k)    # First bucket node
                j = (u, k+1)  # Next bucket node

                # Get dual variables for these nodes
                dual_i = dual_vars.get(f"capacity_flow_conservation_{u}_{k}", 0)
                dual_j = dual_vars.get(f"capacity_flow_conservation_{u}_{k+1}", 0)

                print(f"  Comparing buckets {k} and {k+1}:")
                print(f"    Dual values: {dual_i:.6f} vs {dual_j:.6f}")
```

```python

                # If duals are equal (within numerical tolerance), mark for merging
                if abs(dual_i - dual_j) < 1e-6:
                    print(f"    -> Will merge")
                    buckets_to_merge.append((k, k+1))
                else:
                    print(f"    -> Keep separate")

            # Merge marked buckets (work backwards to avoid index issues)
            for k1, k2 in reversed(buckets_to_merge):
                lower = self.D_u[u][k1][0]  # Lower bound of first bucket
                upper = self.D_u[u][k2][1]  # Upper bound of second bucket
                # Remove the two original buckets and insert merged bucket
                self.D_u[u].pop(k2)
                self.D_u[u].pop(k1)
                self.D_u[u].insert(k1, (lower, upper))

            print(f"  After merge: {self.D_u[u]}")

    def _merge_time_buckets(self, dual_vars):
        """Merge time buckets when their dual variables are equal"""
        print("\nMerging time buckets...")
        for u in self.customers:
            print(f"\nCustomer {u}:")
            print(f"  Before merge: {self.T_u[u]}")

            buckets_to_merge = []

            # Get all consecutive bucket pairs for this customer
            for k in range(len(self.T_u[u]) - 1):
                i = (u, k)    # First bucket node
                j = (u, k+1)  # Next bucket node

                # Get dual variables for these nodes
                dual_i = dual_vars.get(f"time_flow_conservation_{u}_{k}", 0)
                dual_j = dual_vars.get(f"time_flow_conservation_{u}_{k+1}", 0)

                print(f"  Comparing buckets {k} and {k+1}:")
                print(f"    Dual values: {dual_i:.6f} vs {dual_j:.6f}")

                # If duals are equal (within numerical tolerance), mark for merging
                if abs(dual_i - dual_j) < 1e-6:
                    print(f"    -> Will merge")
                    buckets_to_merge.append((k, k+1))
                else:
```

```
                    print(f"    -> Keep separate")

            # Merge marked buckets (work backwards to avoid index issues)
            for k1, k2 in reversed(buckets_to_merge):
                lower = self.T_u[u][k1][0]  # Lower bound of first bucket
                upper = self.T_u[u][k2][1]  # Upper bound of second bucket
                # Remove the two original buckets and insert merged bucket
                self.T_u[u].pop(k2)
                self.T_u[u].pop(k1)
                self.T_u[u].insert(k1, (lower, upper))

            print(f"  After merge: {self.T_u[u]}")

    def _is_significant_flow(self, flow, u_i, u_j):
        """Determine if a flow is significant enough to trigger bucket expansion"""
        # Flow should be significantly non-zero
        if flow < 1e-4:
            return False

        # Skip flows between customers that are too far apart in time
        # (these are less likely to be in optimal solution)
        travel_time = self.costs[u_i, u_j] / 5
        earliest_i = self.time_windows[u_i][0]
        latest_j = self.time_windows[u_j][1]
        if earliest_i + travel_time > latest_j - 10:  # 10 time units buffer
            return False

        # Skip flows between customers that would exceed capacity
        remaining_capacity = self.vehicle_capacity - self.demands[u_i]
        if self.demands[u_j] > remaining_capacity * 0.9:  # 90% threshold
            return False

        # Skip flows that would create very small buckets
        min_bucket_size = (self.vehicle_capacity - min(self.demands.values())) * 0.1  # 10% of max remaining
        if remaining_capacity < min_bucket_size:
            return False

        return True

    def _expand_capacity_buckets(self, z_D):
        """Add new capacity thresholds based on flow solution"""
        print("\nExpanding capacity buckets...")

        # First, create a mapping of flows to actual bucket indices
        flow_mapping = {}
        for (i, j), flow in z_D.items():
```

```
            u_i, k_i = i  # Source node customer and bucket index
            u_j, k_j = j  # Target node customer and bucket index

            # Skip depot nodes and insignificant flows
            if u_j in [self.depot_start, self.depot_end]:
                continue

            if not self._is_significant_flow(flow, u_i, u_j):
                print(f"Skipping insignificant flow ({u_i},{k_i}) -> ({u_j},{k_j}) with flow {flow}")
                continue

            # Find current bucket indices after merging
            source_bucket_idx = None
            target_bucket_idx = None

            # Find which bucket contains the original k_i index's value
            for idx, (lower, upper) in enumerate(self.D_u[u_i]):
                if k_i * ((self.vehicle_capacity - self.demands[u_i]) / 3) <= upper:
                    source_bucket_idx = idx
                    break

            # Find which bucket contains the original k_j index's value
            for idx, (lower, upper) in enumerate(self.D_u[u_j]):
                if k_j * ((self.vehicle_capacity - self.demands[u_j]) / 3) <= upper:
                    target_bucket_idx = idx
                    break

            if source_bucket_idx is not None and target_bucket_idx is not None:
                flow_mapping[(u_i, source_bucket_idx, u_j, target_bucket_idx)] = flow

        # Now process flows with correct bucket indices
        for (u_i, k_i, u_j, k_j), flow in flow_mapping.items():
            print(f"\nProcessing flow ({u_i},{k_i}) -> ({u_j},{k_j}) with flow {flow}")
            print(f"Source customer {u_i} buckets: {self.D_u[u_i]}")
            print(f"Target customer {u_j} buckets: {self.D_u[u_j]}")

            # Calculate new threshold
            d_plus_i = self.D_u[u_i][k_i][1]  # Upper bound of source bucket
            d_u_i = self.demands[u_i]         # Demand of source customer
            new_threshold = d_plus_i - d_u_i

            print(f"  New threshold calculated: {new_threshold}")
```

```
            print(f"  Current buckets for customer {u_j}: {self.D_u[u_j]}")

            # Add new threshold if it's not already present and it's meaningful
            found = False
            for bucket in self.D_u[u_j]:
                if abs(bucket[1] - new_threshold) < 1e-6:
                    found = True
                    print("  -> Threshold already exists")
                    break

            if not found and self.demands[u_j] < new_threshold < self.vehicle_capacity:
                print("  -> Adding new threshold")
                # Insert new bucket maintaining sorted order
                for k, bucket in enumerate(self.D_u[u_j]):
                    if new_threshold < bucket[1]:
                        # Split existing bucket at new threshold
                        if new_threshold > bucket[0]:
                            self.D_u[u_j].insert(k+1, (new_threshold, bucket[1]))
                            self.D_u[u_j][k] = (bucket[0], new_threshold)
                            print(f"  Updated buckets: {self.D_u[u_j]}")
                            break

    def _expand_time_buckets(self, z_T):
        """Add new time thresholds based on flow solution"""
        print("\nExpanding time buckets...")

        # First, create a mapping of flows to actual bucket indices
        flow_mapping = {}
        for (i, j), flow in z_T.items():
            u_i, k_i = i  # Source node customer and bucket index
            u_j, k_j = j  # Target node customer and bucket index

            # Skip depot nodes and insignificant flows
            if u_j in [self.depot_start, self.depot_end]:
                continue

            if not self._is_significant_flow(flow, u_i, u_j):
                print(f"Skipping insignificant flow ({u_i},{k_i}) -> ({u_j},{k_j}) with flow {flow}")
                continue

            # Find current bucket indices after merging
            source_bucket_idx = None
            target_bucket_idx = None

            # Find which bucket contains the original k_i index's value
```

```
            time_span_i = self.time_windows[u_i][1] - self.time_windows[u_i][0]
            for idx, (lower, upper) in enumerate(self.T_u[u_i]):
                if k_i * (time_span_i / 3) <= upper:
                    source_bucket_idx = idx
                    break

            # Find which bucket contains the original k_j index's value
            time_span_j = self.time_windows[u_j][1] - self.time_windows[u_j][0]
            for idx, (lower, upper) in enumerate(self.T_u[u_j]):
                if k_j * (time_span_j / 3) <= upper:
                    target_bucket_idx = idx
                    break

            if source_bucket_idx is not None and target_bucket_idx is not None:
                flow_mapping[(u_i, source_bucket_idx, u_j, target_bucket_idx)] = flow

        # Now process flows with correct bucket indices
        for (u_i, k_i, u_j, k_j), flow in flow_mapping.items():
            print(f"\nProcessing flow ({u_i},{k_i}) -> ({u_j},{k_j}) with flow {flow}")
            print(f"Source customer {u_i} buckets: {self.T_u[u_i]}")
            print(f"Target customer {u_j} buckets: {self.T_u[u_j]}")

            # Calculate new threshold
            t_plus_i = self.T_u[u_i][k_i][1]  # Upper bound of source bucket
            travel_time = self.costs[u_i,u_j] / 5  # Travel time between customers
            t_plus_j = self.T_u[u_j][k_j][1]  # Upper bound of target bucket

            new_threshold = min(t_plus_i - travel_time, t_plus_j)

            print(f"  New threshold calculated: {new_threshold}")
            print(f"  Current buckets for customer {u_j}: {self.T_u[u_j]}")

            # Add new threshold if it's not already present and it's meaningful
            found = False
            for bucket in self.T_u[u_j]:
                if abs(bucket[1] - new_threshold) < 1e-6:
                    found = True
                    print("  -> Threshold already exists")
                    break

```

```
            if not found and self.time_windows[u_j][0] < new_threshold < self.time_windows[u_j][1]:
                print("  -> Adding new threshold")
                # Insert new bucket maintaining sorted order
                for k, bucket in enumerate(self.T_u[u_j]):
                    if new_threshold < bucket[1]:
                        # Split existing bucket at new threshold
                        if new_threshold > bucket[0]:
                            self.T_u[u_j].insert(k+1, (new_threshold, bucket[1]))
                            self.T_u[u_j][k] = (bucket[0], new_threshold)
                            print(f"  Updated buckets: {self.T_u[u_j]}")
                            break

    def _update_bucket_graphs(self):
        """Update time and capacity graphs after bucket modifications"""
        self.nodes_T, self.edges_T = self._create_time_graph()
        self.nodes_D, self.edges_D = self._create_capacity_graph()

        print("\nUpdated graphs:")
        print(f"Time graph: {len(self.nodes_T)} nodes, {len(self.edges_T)} edges")
        print(f"Capacity graph: {len(self.nodes_D)} nodes, {len(self.edges_D)} edges")

    def _reset_neighborhoods(self):
        """Reset LA neighborhoods to maximum size"""
        for u in self.customers:
            distances = [(j, self.costs[u,j]) for j in self.customers if j != u]
            distances.sort(key=lambda x: x[1])
            self.la_neighbors[u] = [j for j, _ in distances[:self.K]]
            print(f"Reset neighbors for customer {u}: {self.la_neighbors[u]}")

    def _validate_buckets(self):
        """Validate bucket structures after modifications"""
        # Check capacity buckets
        for u in self.customers:
            # Verify bucket continuity
            prev_upper = None
            for i, (lower, upper) in enumerate(self.D_u[u]):
                # Check bounds
                if lower >= upper:
                    raise ValueError(f"Invalid capacity bucket bounds for customer {u}: [{lower}, {upper}]")

                # Check ordering
                if prev_upper is not None and abs(lower - prev_upper) >
```

```python
1e-6:
                    raise ValueError(f"Gap in capacity buckets for customer {u} between {prev_upper} and {lower}")

                # Check within vehicle capacity
                if upper > self.vehicle_capacity:
                    raise ValueError(f"Capacity bucket for customer {u} exceeds vehicle capacity: {upper}")

                prev_upper = upper

        # Check time buckets
        for u in self.customers:
            # Verify bucket continuity
            prev_upper = None
            for i, (lower, upper) in enumerate(self.T_u[u]):
                # Check bounds
                if lower >= upper:
                    raise ValueError(f"Invalid time bucket bounds for customer {u}: [{lower}, {upper}]")

                # Check ordering
                if prev_upper is not None and abs(lower - prev_upper) > 1e-6:
                    raise ValueError(f"Gap in time buckets for customer {u} between {prev_upper} and {lower}")

                # Check within time windows
                if upper > self.time_windows[u][1] or lower < self.time_windows[u][0]:
                    raise ValueError(f"Time bucket for customer {u} outside time window: [{lower}, {upper}]")

                prev_upper = upper

def load_solomon_instance(filename, customer_ids=None):
    """
    Load Solomon VRPTW instance from CSV file

    Args:
        filename: Path to CSV file
        customer_ids: List of specific customer IDs to include (None for all)
    """
    # Read CSV file
    df = pd.read_csv(filename)

    # Convert 'Depot' to 0 in CUST_NUM
    df['CUST_NUM'] = df['CUST_NUM'].replace('Depot', '0')
```

```
    df['CUST_NUM'] = df['CUST_NUM'].astype(int)

    # Filter customers if specific IDs provided
    if customer_ids is not None:
        selected_ids = [0] + sorted(customer_ids)
        df = df[df['CUST_NUM'].isin(selected_ids)]

        if len(df) != len(selected_ids):
            missing = set(selected_ids) - set(df['CUST_NUM'])
            raise ValueError(f"Customer IDs not found: {missing}")

    # Extract coordinates
    coords = {row['CUST_NUM']: (row['XCOORD.'], row['YCOORD.'])
             for _, row in df.iterrows()}

    # Create customer list (excluding depot)
    customers = sorted(list(set(df['CUST_NUM']) - {0}))

    # Add virtual end depot with same coordinates as start depot
    virtual_end = max(customers) + 1
    coords[virtual_end] = coords[0]

    # Calculate costs/distances (not including service times)
    costs = {}
    all_nodes = [0] + customers + [virtual_end]
    for i in all_nodes:
        for j in all_nodes:
            if i != j:
                x1, y1 = coords[i]
                x2, y2 = coords[j]
                costs[i,j] = np.floor(np.sqrt((x2-x1)**2 + (y2-y1)**2) * 
10) / 10

    # Extract service times (0 for depots)
    service_times = {row['CUST_NUM']: row['SERVICE_TIME']
                    for _, row in df.iterrows()}
    service_times[virtual_end] = 0  # No service time at end depot

    # Extract time windows and demands
    time_windows = {row['CUST_NUM']: (row['READY_TIME'], row['DUE_DATE'])
                   for _, row in df.iterrows()}
    time_windows[virtual_end] = time_windows[0]  # End depot has same time 
window as start

    demands = {row['CUST_NUM']: row['DEMAND']
              for _, row in df.iterrows()}
    demands[virtual_end] = 0

    # Create instance dictionary
```

```
    instance = {
        'customers': customers,
        'depot_start': 0,
        'depot_end': virtual_end,
        'costs': costs,  # Just distances between customers
        'time_windows': time_windows,
        'service_times': service_times,  # Added service times
        'demands': demands,
        'vehicle_capacity': 200
    }

    print("\nSelected Customer Details:")
    print("ID  Ready  Due    Service  Demand  Location")
    print("-" * 50)
    for c in customers:
        x, y = coords[c]
        tw = time_windows[c]
        print(f"{c:<3} {tw[0]:<6} {tw[1]:<6} {service_times[c]:<8} {demands[c]:<7} ({x},{y})")

    return instance

def run_solomon_instance(filename, customer_ids, K=3, time_granularity=3, capacity_granularity=3, max_iterations=5, time_limit=300):
    print(f"Loading Solomon instance with {len(customer_ids)} selected customers...")
    instance = load_solomon_instance(filename, customer_ids)

    optimizer = VRPTWOptimizer(
        customers=instance['customers'],
        depot_start=instance['depot_start'],
        depot_end=instance['depot_end'],
        costs=instance['costs'],
        time_windows=instance['time_windows'],
        service_times=instance['service_times'],  # Added service times
        demands=instance['demands'],
        vehicle_capacity=instance['vehicle_capacity'],
        K=K,
        time_granularity=time_granularity,
        capacity_granularity=capacity_granularity,
        max_iterations=max_iterations
    )

    print("\nSolving...")
    solution = optimizer.solve_with_parsimony(time_limit=time_limit)

    return optimizer, solution

def test_phi_functions():
```

```
    """Test the LA Arc computation with multiple test cases"""
    # Test case data
    time_windows = {
        1: (10, 40),    # Customer 1
        2: (30, 70),    # Customer 2
        3: (50, 80),    # Customer 3
        4: (20, 60),    # Customer 4
        5: (40, 90),    # Customer 5
    }

    # Customer locations
    locations = {
        1: (2, 4),      # Customer 1
        2: (-1, 3),     # Customer 2
        3: (4, 1),      # Customer 3
        4: (-2, -3),    # Customer 4
        5: (1, -2),     # Customer 5
    }

    # Calculate all travel times
    travel_times = {}
    for i in locations:
        for j in locations:
            if i != j:
                x1, y1 = locations[i]
                x2, y2 = locations[j]
                dist = np.floor(np.sqrt((x2-x1)**2 + (y2-y1)**2) * 10) / 10
                travel_times[(i,j)] = dist

    la_computer = LAArcComputer(time_windows, travel_times)

    # Test Case 1: Simple two-customer sequence
    print("\nTest Case 1: Two customers [1,2]")
    r1 = [1, 2]
    phi_r1 = la_computer.compute_phi_r(r1)
    phi_hat_r1 = la_computer.compute_phi_hat_r(r1)
    print(f"φ_r: {phi_r1}, φ̂_r: {phi_hat_r1}")
    # The earliest departure should be earlier than the latest departure
    assert phi_r1 <= phi_hat_r1, "Earliest departure should be before latest departure"
    # Both values should be within the first customer's time window
    assert time_windows[1][0] <= phi_r1 <= time_windows[1][1], "φ_r outside time window"
    assert time_windows[1][0] <= phi_hat_r1 <= time_windows[1][1], "φ̂_r outside time window"

    # Test Case 2: Three-customer sequence
    print("\nTest Case 2: Three customers [1,2,3]")
    r2 = [1, 2, 3]
```

```
    phi_r2 = la_computer.compute_phi_r(r2)
    phi_hat_r2 = la_computer.compute_phi_hat_r(r2)
    print(f"φ_r: {phi_r2}, φ̂_r: {phi_hat_r2}")
    assert phi_r2 <= phi_hat_r2

    # Test Case 3: Different three-customer sequence
    print("\nTest Case 3: Three customers [4,5,1]")
    r3 = [4, 5, 1]
    phi_r3 = la_computer.compute_phi_r(r3)
    phi_hat_r3 = la_computer.compute_phi_hat_r(r3)
    print(f"φ_r: {phi_r3}, φ̂_r: {phi_hat_r3}")
    assert phi_r3 <= phi_hat_r3

    # Detailed analysis of results
    print("\nDetailed Analysis:")
    print("Travel times between customers:")
    for (i,j), time in sorted(travel_times.items()):
        print(f"From {i} to {j}: {time}")

    print("\nTime windows:")
    for i, window in sorted(time_windows.items()):
        print(f"Customer {i}: {window}")

'''if __name__ == "__main__":
    print("Testing LA Arc computation...")
    test_phi_functions()'''

if __name__ == "__main__":
    optimizer, solution = run_solomon_instance(
        filename="r109.csv",
        # customer_ids=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25],
        customer_ids=list(range(1,51)),
        K=10,
        time_granularity=10,
        capacity_granularity=10,
        max_iterations=5,
        time_limit=300
    )
```