

```
1 import gurobipy as gp
2 from gurobipy import GRB
3 import numpy as np
4 import time
5 from collections import defaultdict
6 import pandas as pd
7
8 class LAArcComputer:
9     def __init__(self, time_windows, travel_times):
10         """
11         Initialize LA Arc Computer
12
13         Args:
14             time_windows: Dict[int, Tuple[float, float]] - For each
... customer, (earliest, latest) service times
15             travel_times: Dict[Tuple[int, int], float] - Travel time
... between each pair of customers
16         """
17         self.time_windows = time_windows
18         self.travel_times = travel_times
19
20     def compute_phi_r(self, r: list) -> float:
21         """
22         Compute  $\phi_r$ : earliest time we could leave first customer without
... waiting
23         """
24         print(f"\nComputing  $\phi_r$  for sequence {r}")
25         if len(r) == 2: # Base case (13a)
26             u, v = r
27             t_minus_u = self.time_windows[u][0]
28             t_minus_v = self.time_windows[v][0]
29             t_uv = self.travel_times[u,v]
30             result = max(t_minus_u, t_minus_v - t_uv)
31             print(f"Base case  $\phi_r$ :")
32             print(f"  Customer {u} earliest time (t_minus_u): {t_minus_u}")
33             print(f"  Customer {v} earliest time (t_minus_v): {t_minus_v}")
34             print(f"  Travel time (t_uv): {t_uv}")
35             print(f"  Result: max({t_minus_u}, {t_minus_v} - {t_uv}) =
... {result}")
36             return result
37
38         else: # Recursive case (13c)
39             u = r[0]
40             w = r[1]
41             t_minus_u = self.time_windows[u][0]
42             t_uw = self.travel_times[u,w]
43             print(f"Recursive case for {r}:")
44             print(f"  Computing  $\phi_r$  for subsequence {r[1:]}")
45             phi_r_minus = self.compute_phi_r(r[1:])
```

```
46         result = max(t_minus_u, phi_r_minus - t_uw) # Propagate time
... backwards
47         print(f"    Customer {u} earliest time (t_minus_u): {t_minus_u}")
48         print(f"    Travel time to {w} (t_uw): {t_uw}")
49         print(f"    Recursive result (phi_r_minus): {phi_r_minus}")
50         print(f"    Final result: max({t_minus_u}, {phi_r_minus} -
... {t_uw}) = {result}")
51         return result
52
53     def compute_phi_hat_r(self, r: list) -> float:
54         """
55         Compute  $\hat{\phi}_r$ : latest feasible departure time from first customer
56         """
57         print(f"\nComputing  $\hat{\phi}_r$  for sequence {r}")
58         if len(r) == 2: # Base case (13b)
59             u, v = r
60             t_plus_u = self.time_windows[u][1]
61             t_minus_v = self.time_windows[v][0] # Need earliest time at v
62             t_uv = self.travel_times[u,v]
63             result = min(t_plus_u, t_minus_v - t_uv)
64             print(f"Base case  $\hat{\phi}_r$ :")
65             print(f"    Customer {u} latest time (t_plus_u): {t_plus_u}")
66             print(f"    Customer {v} earliest time (t_minus_v): {t_minus_v}")
67             print(f"    Travel time (t_uv): {t_uv}")
68             print(f"    Result: min({t_plus_u}, {t_minus_v} - {t_uv}) =
... {result}")
69             return result
70
71         else: # Recursive case (13d)
72             u = r[0]
73             w = r[1]
74             t_plus_u = self.time_windows[u][1]
75             t_uw = self.travel_times[u,w]
76             print(f"Recursive case for {r}:")
77             print(f"    Computing  $\hat{\phi}_r$  for subsequence {r[1:]}")
78             phi_hat_r_minus = self.compute_phi_hat_r(r[1:])
79             result = min(t_plus_u, phi_hat_r_minus - t_uw) # Propagate
... time backwards
80             print(f"    Customer {u} latest time (t_plus_u): {t_plus_u}")
81             print(f"    Travel time to {w} (t_uw): {t_uw}")
82             print(f"    Recursive result (phi_hat_r_minus):
... {phi_hat_r_minus}")
83             print(f"    Final result: min({t_plus_u}, {phi_hat_r_minus} -
... {t_uw}) = {result}")
84             return result
85
86     class VRPTWOptimizer:
87         def __init__(self, customers, depot_start, depot_end, costs,
... time_windows, service_times, demands,
```

```
88         vehicle_capacity, K=3, time_granularity=3,
... capacity_granularity=3, max_iterations=5):
89     self.customers = customers
90     self.depot_start = depot_start
91     self.depot_end = depot_end
92     self.costs = costs
93     self.time_windows = time_windows
94     self.service_times = service_times # Added service times
95     self.demands = demands
96     self.vehicle_capacity = vehicle_capacity
97     self.K = K
98     self.time_granularity = time_granularity
99     self.capacity_granularity = capacity_granularity
100     self.max_iterations = max_iterations
101
102     # Algorithm parameters remain the same
103     self.MIN_INC = 1
104     self.sigma = 9
105
106     self.model = None
107     self.create_initial_model()
108
109     def _add_constraints(self):
110         """Add all constraints"""
111         # Objective function remains unchanged – uses costs (distances)
... only
112         self.model.setObjective(
113             gp.quicksum(self.costs[i,j] * self.x[i,j] for i,j in
... self.E_star),
114             GRB.MINIMIZE
115         )
116
117         # Visit each customer once constraints remain unchanged
118         for u in self.customers:
119             self.model.addConstr(
120                 gp.quicksum(self.x[i,u] for i,j in self.E_star if j == u)
... == 1,
121                 name=f'visit_in_{u}'
122             )
123             self.model.addConstr(
124                 gp.quicksum(self.x[u,j] for i,j in self.E_star if i == u)
... == 1,
125                 name=f'visit_out_{u}'
126             )
127
128         # Time window constraints updated to include service times
129         M = max(tw[1] for tw in self.time_windows.values())
130         for (i,j) in self.E_star:
131             if j != self.depot_end:
```

```
132         self.model.addConstr(
133             self.tau[j] >= self.tau[i] + self.service_times[i] +
... self.costs[i,j]/5
134             - M * (1 - self.x[i,j]),
135             name=f'time_prop_{i}_{j}'
136         )
137
138     # Time window bounds remain unchanged
139     for i in self.customers + [self.depot_start, self.depot_end]:
140         self.model.addConstr(
141             self.tau[i] >= self.time_windows[i][0],
142             name=f'tw_lb_{i}'
143         )
144         self.model.addConstr(
145             self.tau[i] <= self.time_windows[i][1],
146             name=f'tw_ub_{i}'
147         )
148
149     # Other constraints remain unchanged
150     self._add_la_arc_constraints_with_parsimony()
151     self._add_capacity_constraints()
152     self._add_time_flow_constraints()
153     self._add_capacity_flow_constraints()
154     self.model.update()
155
156     def create_initial_model(self):
157         """Create initial model with all components"""
158         self.model = gp.Model("VRPTW")
159
160         # Create valid edges
161         self.E_star = [(i,j) for i in [self.depot_start] + self.customers
162                        for j in self.customers + [self.depot_end] if i != j]
163
164         # Generate LA neighborhoods
165         self.la_neighbors = self._generate_initial_la_neighbors()
166
167         # Generate orderings
168         self.R_u = self._generate_orderings()
169
170         # Create time and capacity discretization
171         self.T_u = self._create_time_buckets()
172         self.D_u = self._create_capacity_buckets()
173
174         # Create flow graphs first
175         self.nodes_T, self.edges_T = self._create_time_graph()
176         self.nodes_D, self.edges_D = self._create_capacity_graph()
177
178         # Then create variables that depend on the graphs
179         self._create_variables()
```

```
180
181     # Finally add constraints
182     self._add_constraints()
183
184     def _create_variables(self):
185         """Create all decision variables"""
186         # Route variables x_{ij}
187         self.x = {}
188         for i, j in self.E_star:
189             self.x[i,j] = self.model.addVar(vtype=GRB.BINARY,
190 ... name=f'x_{i}_{j}')
191
192         # Time variables  $\tau_i$ 
193         self.tau = {}
194         for i in self.customers + [self.depot_start, self.depot_end]:
195             self.tau[i] = self.model.addVar(lb=0, name=f'tau_{i}')
196
197         # LA-arc variables y_r
198         self.y = {}
199         for u in self.customers:
200             for r in range(len(self.R_u[u])):
201                 self.y[u,r] = self.model.addVar(vtype=GRB.BINARY,
202 ... name=f'y_{u}_{r}')
203
204         # Time flow variables z_T
205         self.z_T = {}
206         for edge in self.edges_T:
207             self.z_T[edge] = self.model.addVar(lb=0, name=f'z_T_{edge}')
208
209         # Capacity flow variables z_D
210         self.z_D = {}
211         for edge in self.edges_D:
212             self.z_D[edge] = self.model.addVar(lb=0, name=f'z_D_{edge}')
213
214         # Load tracking variables
215         self.load = {}
216         for i in self.customers:
217             self.load[i] = self.model.addVar(lb=0,
218 ... ub=self.vehicle_capacity, name=f'load_{i}')
219
220         self.model.update()
221
222         def _add_la_arc_constraints_with_parsimony(self):
223             """Add LA-arc movement consistency constraints with parsimony
224 ... penalties"""
225             # Small positive value for parsimony penalties
226             rho = 0.01 #  $\rho$  in the paper
227
228             # For each customer u, track customers by distance
```

```
225         for u in self.customers:
226             # Get distances to all other customers
227             distances = [(j, self.costs[u,j]) for j in self.customers if j
... != u]
228             distances.sort(key=lambda x: x[1])
229
230             # Create k-indexed neighborhoods ( $N^k_u$ )
231             N_k_u = {} # k-indexed neighborhoods
232             N_k_plus_u = {} #  $N^k_u$  from paper (includes u)
233             for k in range(1, len(distances) + 1):
234                 N_k_u[k] = [j for j, _ in distances[:k]]
235                 N_k_plus_u[k] = [u] + N_k_u[k]
236
237             # Select one ordering per customer (equation 6g)
238             self.model.addConstr(
239                 gp.quicksum(self.y[u,r] for r in range(len(self.R_u[u])))
... == 1,
240                 name=f'one_ordering_{u}'
241             )
242
243             # Add k-indexed constraints from equation (8a)
244             for k in range(1, len(distances) + 1):
245                 for w in N_k_plus_u[k]:
246                     for v in N_k_u[k]:
247                         if (w,v) in self.E_star:
248                             self.model.addConstr(
249                                 rho * k + self.x[w,v] >= gp.quicksum(
250                                     self.y[u,r] for r in
... range(len(self.R_u[u]))
251                                     if
... self._is_in_k_neighborhood_ordering(r, w, v, k, N_k_u[k])
252                                     ),
253                                     name=f'la_arc_cons_{u}_{w}_{v}_{k}'
254                                 )
255
256             # Add k-indexed constraints from equation (8b)
257             for k in range(1, len(distances) + 1):
258                 for w in N_k_plus_u[k]:
259                     outside_neighbors = [j for j in self.customers +
... [self.depot_end]
260                                         if j not in N_k_plus_u[k]]
261                     if outside_neighbors:
262                         self.model.addConstr(
263                             rho * k + gp.quicksum(
264                                 self.x[w,j] for j in outside_neighbors
265                                 if (w,j) in self.E_star
266                             ) >= gp.quicksum(
267                                 self.y[u,r] for r in
... range(len(self.R_u[u]))
```

```
268         if self._is_final_in_k_neighborhood(r, w,
... k, N_k_plus_u[k])
269             ),
270             name=f'la_arc_final_{u}_{w}_{k}'
271         )
272
273     def _add_capacity_constraints(self):
274         """Add capacity flow constraints"""
275         # Initial load from depot
276         for j in self.customers:
277             self.model.addConstr(
278                 self.load[j] >= self.demands[j] *
... self.x[self.depot_start, j],
279                 name=f'init_load_{j}'
280             )
281
282         # Load propagation between customers
283         M = self.vehicle_capacity
284         for i in self.customers:
285             for j in self.customers:
286                 if i != j and (i, j) in self.E_star:
287                     self.model.addConstr(
288                         self.load[j] >= self.load[i] + self.demands[j] - M
... * (1 - self.x[i, j]),
289                         name=f'load_prop_{i}_{j}'
290                     )
291
292         # Enforce capacity limit
293         for i in self.customers:
294             self.model.addConstr(
295                 self.load[i] <= self.vehicle_capacity,
296                 name=f'cap_limit_{i}'
297             )
298
299     def _add_capacity_flow_constraints(self):
300         """Add capacity flow constraints (equations 4a and 4b from
... paper)"""
301         # Flow conservation (4a)
302         for i, k, d_min, d_max in self.nodes_D:
303             if i not in [self.depot_start, self.depot_end]:
304                 self.model.addConstr(
305                     gp.quicksum(self.z_D[e] for e in self.edges_D if e[0]
... == (i, k)) ==
306                     gp.quicksum(self.z_D[e] for e in self.edges_D if e[1]
... == (i, k)),
307                     name=f'cap_flow_cons_{i}_{k}'
308                 )
309
310         # Consistency with route variables (4b)
```

```
311         for u,v in self.E_star:
312             self.model.addConstr(
313                 self.x[u,v] == gp.quicksum(
314                     self.z_D[e] for e in self.edges_D
315                     if e[0][0] == u and e[1][0] == v
316                 ),
317                 name=f'cap_flow_cons_route_{u}_{v}'
318             )
319
320     def _add_time_flow_constraints(self):
321         """Add time flow constraints"""
322         # Flow conservation (5a)
323         for i, k, t_min, t_max in self.nodes_T:
324             if i not in [self.depot_start, self.depot_end]:
325                 self.model.addConstr(
326                     gp.quicksum(self.z_T[e] for e in self.edges_T if e[0]
... == (i,k)) ==
327                     gp.quicksum(self.z_T[e] for e in self.edges_T if e[1]
... == (i,k)),
328                     name=f'time_flow_cons_{i}_{k}'
329                 )
330
331         # Consistency with route variables (5b)
332         for u,v in self.E_star:
333             self.model.addConstr(
334                 self.x[u,v] == gp.quicksum(
335                     self.z_T[e] for e in self.edges_T
336                     if e[0][0] == u and e[1][0] == v
337                 ),
338                 name=f'time_flow_cons_route_{u}_{v}'
339             )
340
341         # Link time variables  $\tau$  with time buckets
342         M = max(tw[1] for tw in self.time_windows.values())
343         for i, k, t_min, t_max in self.nodes_T:
344             if i not in [self.depot_start, self.depot_end]:
345                 outgoing_edges = [e for e in self.edges_T if e[0] == (i,k)]
346                 if outgoing_edges:
347                     self.model.addConstr(
348                         self.tau[i] >= t_min - M * (1 -
... gp.quicksum(self.z_T[e] for e in outgoing_edges)),
349                         name=f'time_bucket_lb_{i}_{k}'
350                     )
351                     self.model.addConstr(
352                         self.tau[i] <= t_max + M * (1 -
... gp.quicksum(self.z_T[e] for e in outgoing_edges)),
353                         name=f'time_bucket_ub_{i}_{k}'
354                     )
355
```



```
356     def solve_with_parsimony(self, time_limit=None):
357         """Solve VRPTW with LA neighborhood parsimony"""
358         print("\n=== Initial LA-Neighborhood Analysis ===")
359         self._print_neighborhood_analysis()
360
361         if time_limit:
362             self.model.setParam('TimeLimit', time_limit)
363
364         # Set other Gurobi parameters
365         self.model.setParam('MIPGap', 0.01) # 1% optimality gap
366         self.model.setParam('Threads', 4) # Use 4 threads
367
368         iteration = 1
369         last_lp_val = float('-inf')
370         iter_since_reset = 0
371
372         while iteration <= self.max_iterations:
373             print(f"\n=== Iteration {iteration} ===")
374
375             # Solve current iteration
376             self.model.optimize()
377
378             current_obj = self.model.objVal if self.model.Status in
... [GRB.OPTIMAL, GRB.TIME_LIMIT] else None
379
380             print(f"\nIteration Objective: {current_obj}")
381
382             if current_obj is not None and current_obj > last_lp_val +
... self.MIN_INC:
383                 print("Solution improved")
384                 last_lp_val = current_obj
385                 iter_since_reset = 0
386
387                 try:
388                     # Get dual variables and analyze buckets
389                     print("\n--- Getting LP Relaxation Information ---")
390                     dual_vars = self.get_dual_variables()
391
392                     if dual_vars: # Only analyze if we got dual variables
393                         print("\n--- LA-Neighborhood Analysis After
... Improvement ---")
394                         self._print_neighborhood_analysis()
395                         print("\n--- Bucket Analysis ---")
396                         self._print_bucket_analysis()
397                     except Exception as e:
398                         print(f"Warning: Could not complete analysis due to:
... {e}")
399
400                 else:
```

```
401         print("No significant improvement")
402         iter_since_reset += 1
403
404     if iter_since_reset >= self.sigma:
405         print("\nResetting neighborhoods to maximum size...")
406         self._reset_neighborhoods()
407         iter_since_reset = 0
408
409     iteration += 1
410
411     # Extract final solution
412     solution = self._extract_solution()
413     if solution['status'] == 'Optimal':
414         self.validate_solution(solution)
415
416     return solution
417
418 def _print_neighborhood_analysis(self):
419     """Print detailed analysis of LA neighborhoods"""
420     print("\nLA-Neighborhood Details:")
421
422     # Print sizes and members
423     for u in self.customers:
424         neighbors = self.la_neighbors[u]
425         print(f"\nCustomer {u}:")
426         print(f"  Neighborhood size: {len(neighbors)}")
427         print(f"  Neighbors: {neighbors}")
428
429     # Print distances to neighbors
430     distances = [(j, self.costs[u,j]) for j in neighbors]
431     distances.sort(key=lambda x: x[1])
432     print("  Distances to neighbors:")
433     for j, dist in distances:
434         print(f"    -> Customer {j}: {dist/5:.1f}")
435
436     # Print time window compatibility
437     print("  Time window compatibility:")
438     for j in neighbors:
439         u_early, u_late = self.time_windows[u]
440         j_early, j_late = self.time_windows[j]
441         travel_time = self.costs[u,j]/5
442         print(f"    -> Customer {j}: Window [{j_early}, {j_late}]")
443         print(f"    Earliest possible arrival: {u_early +
... travel_time:.1f}")
444         print(f"    Latest possible arrival: {u_late +
... travel_time:.1f}")
445
446 def _print_bucket_analysis(self):
447     """Print detailed analysis of time and capacity buckets"""
```

```
448     print("\nTime Bucket Analysis:")
449     for u in self.customers:
450         print(f"\nCustomer {u}:")
451         print(f"    Time window: {self.time_windows[u]}")
452         print(f"    Number of buckets: {len(self.T_u[u])}")
453         print("    Bucket ranges:")
454         for i, (t_min, t_max) in enumerate(self.T_u[u]):
455             print(f"        Bucket {i}: [{t_min:.1f}, {t_max:.1f}]")
456
457     print("\nCapacity Bucket Analysis:")
458     for u in self.customers:
459         print(f"\nCustomer {u}:")
460         print(f"    Demand: {self.demands[u]}")
461         print(f"    Number of buckets: {len(self.D_u[u])}")
462         print("    Bucket ranges:")
463         for i, (d_min, d_max) in enumerate(self.D_u[u]):
464             print(f"        Bucket {i}: [{d_min:.1f}, {d_max:.1f}]")
465
466     def _extract_solution(self):
467         """Extract solution details"""
468         if self.model.Status == GRB.OPTIMAL:
469             status = 'Optimal'
470         elif self.model.Status == GRB.TIME_LIMIT:
471             status = 'TimeLimit'
472         else:
473             status = 'Other'
474
475         solution = {
476             'status': status,
477             'objective': self.model.ObjVal if status in ['Optimal',
478 ... 'TimeLimit'] else None,
479             'routes': self._extract_routes() if status in ['Optimal',
480 ... 'TimeLimit'] else None,
481             'computation_time': self.model.Runtime,
482             'neighborhood_sizes': {u: len(neighbors) for u, neighbors in
483 ... self.la_neighbors.items()}
484         }
485
486         return solution
487
488     def _extract_routes(self):
489         """Extract routes from solution"""
490         if self.model.Status not in [GRB.OPTIMAL, GRB.TIME_LIMIT]:
491             return None
492
493         active_edges = [(i,j) for (i,j) in self.E_star
494 ... if self.x[i,j].X > 0.5]
495
496         routes = []
```

```
494         depot_starts = [(i,j) for (i,j) in active_edges if i ==
... self.depot_start]
495
496         for start_edge in depot_starts:
497             route = []
498             current = start_edge[1]
499             route.append(current)
500
501             while current != self.depot_end:
502                 next_edges = [(i,j) for (i,j) in active_edges if i ==
... current]
503
504                 if not next_edges:
505                     break
506                 current = next_edges[0][1]
507                 if current != self.depot_end:
508                     route.append(current)
509
510             routes.append(route)
511
512         return routes
513
514     def _generate_initial_la_neighbors(self):
515         """Generate initial LA neighborhoods using K closest neighbors"""
516         la_neighbors = {}
517         for u in self.customers:
518             # Get distances to all other customers
519             distances = [(j, self.costs[u,j]) for j in self.customers if j
... != u]
520
521             distances.sort(key=lambda x: x[1])
522
523             # Take K closest neighbors that are reachable
524             neighbors = []
525             for j, _ in distances:
526                 if len(neighbors) >= self.K:
527                     break
528                 if self._is_reachable(u, j):
529                     neighbors.append(j)
530             la_neighbors[u] = neighbors
531         return la_neighbors
532
533     def _is_reachable(self, i, j):
534         """Check if j is reachable from i considering time windows and
... capacity"""
535         earliest_i, latest_i = self.time_windows[i]
536         earliest_j, latest_j = self.time_windows[j]
537         travel_time = self.costs[i,j] / 5
538
539         if earliest_i + travel_time > latest_j:
540             return False
```

```
539
540     if self.demands[i] + self.demands[j] > self.vehicle_capacity:
541         return False
542
543     return True
544
545     def _generate_orderings(self):
546         """Generate efficient orderings for each customer"""
547         R_u = defaultdict(list)
548
549         for u in self.customers:
550             # Base ordering
551             R_u[u].append({
552                 'sequence': [u],
553                 'a_wv': {},
554                 'a_star': {u: 1}
555             })
556
557             # Add single neighbor orderings
558             for v in self.la_neighbors[u]:
559                 if self._is_reachable(u, v):
560                     R_u[u].append({
561                         'sequence': [u, v],
562                         'a_wv': {(u,v): 1},
563                         'a_star': {v: 1}
564                     })
565
566             # Add two-neighbor orderings
567             for v1 in self.la_neighbors[u]:
568                 for v2 in self.la_neighbors[u]:
569                     if v1 != v2 and self._is_sequence_feasible([u, v1,
... v2]):
570                         R_u[u].append({
571                             'sequence': [u, v1, v2],
572                             'a_wv': {(u,v1): 1, (v1,v2): 1},
573                             'a_star': {v2: 1}
574                         })
575
576             return R_u
577
578     def _is_sequence_feasible(self, sequence):
579         """Check if a sequence of customers is feasible, updated to include
... service times"""
580         # Check capacity constraints
581         total_demand = sum(self.demands[i] for i in sequence)
582         if total_demand > self.vehicle_capacity:
583             return False
584
585         # Check time feasibility including service times
```

```
586         current_time = self.time_windows[sequence[0]][0] # Start at
... earliest possible time
587
588         for i in range(len(sequence)):
589             current = sequence[i]
590
591             # Cannot arrive after latest time window
592             if current_time > self.time_windows[current][1]:
593                 return False
594
595             # Update time to include service
596             current_time = max(current_time, self.time_windows[current][0])
597             current_time += self.service_times[current]
598
599             # Add travel time to next customer if any
600             if i < len(sequence)-1:
601                 next_customer = sequence[i+1]
602                 current_time += self.costs[current,next_customer]/5
603
604         return True
605
606     def _is_in_k_neighborhood_ordering(self, r, w, v, k, N_k_u):
607         """Check if w immediately precedes v in ordering r within
... k-neighborhood"""
608         for u in self.R_u:
609             if r < len(self.R_u[u]):
610                 ordering = self.R_u[u][r]
611                 sequence = ordering['sequence']
612
613                 # Both customers must be in k-neighborhood
614                 if w not in N_k_u or v not in N_k_u:
615                     return False
616
617                 # Check if w immediately precedes v
618                 for i in range(len(sequence)-1):
619                     if sequence[i] == w and sequence[i+1] == v:
620                         return True
621                 return False
622         return False
623
624     def _is_final_in_k_neighborhood(self, r, w, k, N_k_plus_u):
625         """Check if w is final customer in k-neighborhood for ordering r"""
626         for u in self.R_u:
627             if r < len(self.R_u[u]):
628                 ordering = self.R_u[u][r]
629                 sequence = ordering['sequence']
630
631                 # w must be in k-neighborhood
632                 if w not in N_k_plus_u:
```

```
633         return False
634
635         # Find position of w in sequence
636         try:
637             w_pos = sequence.index(w)
638         except ValueError:
639             return False
640
641         # Check if w is last in sequence or followed by customer
... outside k-neighborhood
642         return (w_pos == len(sequence)-1 or
643                 sequence[w_pos+1] not in N_k_plus_u)
644     return False
645
646     def validate_solution(self, solution):
647         """Validate solution feasibility, updated to include service
... times"""
648         if solution['status'] != 'Optimal':
649             return False
650
651         routes = solution['routes']
652         print("\nValidating solution:")
653
654         for idx, route in enumerate(routes, 1):
655             print(f"\nRoute {idx}: {' -> '.join(map(str, [self.depot_start]
... + route + [self.depot_end]))}")
656
657             # Check capacity
658             route_load = sum(self.demands[i] for i in route)
659             print(f"  Load: {route_load}/{self.vehicle_capacity}", end=" ")
660             if route_load > self.vehicle_capacity:
661                 print("✗ Exceeds capacity!")
662                 return False
663             print("✓")
664
665             # Check time windows including service times
666             current_time = 0
667             current_loc = self.depot_start
668             for stop in route:
669                 travel_time = self.costs[current_loc, stop] / 5
670                 arrival_time = current_time + travel_time
671                 service_start = max(arrival_time,
... self.time_windows[stop][0])
672
673                 window_start, window_end = self.time_windows[stop]
674                 print(f"  Customer {stop}:")
675                 print(f"    Arrive: {arrival_time:.1f}")
676                 print(f"    Service start: {service_start:.1f}")
677                 print(f"    Window: [{window_start}, {window_end}]", end="
```

```
677... ")
678
679         if service_start > window_end:
680             print("✗ Misses window!")
681             return False
682         print("✓")
683
684         # Update current time to include service
685         current_time = service_start + self.service_times[stop]
686         current_loc = stop
687
688         return True
689
690     def get_dual_variables(self):
691         """Get dual variables from the LP relaxation"""
692         # For Gurobi, we need to solve the LP relaxation first
693         relaxed = self.model.copy()
694         for v in relaxed.getVars():
695             if v.vType != GRB.CONTINUOUS:
696                 v.vType = GRB.CONTINUOUS
697
698         relaxed.optimize()
699
700         # Get dual values using Pi attribute
701         dual_vars = {}
702         if relaxed.Status == GRB.OPTIMAL:
703             for c in relaxed.getConstrs():
704                 try:
705                     dual_vars[c.ConstrName] =
... relaxed.getConstrByName(c.ConstrName).Pi
706                 except Exception:
707                     # If we can't get the dual for a constraint, skip it
708                     continue
709
710         return dual_vars
711
712     def _create_capacity_buckets(self):
713         """Create initial capacity buckets for each customer"""
714         buckets = {}
715
716         # Create buckets for each customer
717         for u in self.customers:
718             demand = self.demands[u]
719             remaining_capacity = self.vehicle_capacity - demand
720             bucket_size = remaining_capacity / self.capacity_granularity
721
722             customer_buckets = []
723             current_capacity = demand
724
```



```
725         # Create evenly spaced buckets
726         for i in range(self.capacity_granularity):
727             lower = current_capacity
728             upper = min(current_capacity + bucket_size,
... self.vehicle_capacity)
729             customer_buckets.append((lower, upper))
730             current_capacity = upper
731
732             if current_capacity >= self.vehicle_capacity:
733                 break
734
735         buckets[u] = customer_buckets
736
737         # Add single bucket for depot (start and end)
738         buckets[self.depot_start] = [(0, self.vehicle_capacity)]
739         buckets[self.depot_end] = [(0, self.vehicle_capacity)]
740
741         return buckets
742
743     def _create_capacity_graph(self):
744         """Create directed graph GD for capacity flow"""
745         nodes_D = [] # (u, k, d-, d+)
746         edges_D = [] # ((u1,k1), (u2,k2))
747
748         # Create nodes for each customer and their capacity buckets
749         for u in self.customers:
750             for k, (d_min, d_max) in enumerate(self.D_u[u]):
751                 nodes_D.append((u, k, d_min, d_max))
752
753         # Add depot nodes
754         depot_start_bucket = self.D_u[self.depot_start][0]
755         depot_end_bucket = self.D_u[self.depot_end][0]
756         nodes_D.append((self.depot_start, 0, depot_start_bucket[0],
... depot_start_bucket[1]))
757         nodes_D.append((self.depot_end, 0, depot_end_bucket[0],
... depot_end_bucket[1]))
758
759         # Create edges between nodes
760         for i, k_i, d_min_i, d_max_i in nodes_D:
761             for j, k_j, d_min_j, d_max_j in nodes_D:
762                 if i != j:
763                     # Check if edge is feasible based on capacity
764                     demand_j = self.demands[j]
765                     remaining_i = d_max_i - self.demands[i]
766
767                     # Edge is feasible if remaining capacity after i can
... accommodate j
768                     if (remaining_i >= demand_j and
769                         d_min_j >= demand_j and
```

```
770         d_max_j <= self.vehicle_capacity):
771             edges_D.append(((i,k_i), (j,k_j)))
772
773     return nodes_D, edges_D
774
775     def _create_time_buckets(self):
776         """Create initial time buckets for each customer"""
777         buckets = {}
778
779         # Create buckets for each customer
780         for u in self.customers:
781             earliest_time, latest_time = self.time_windows[u]
782             time_span = latest_time - earliest_time
783             bucket_size = time_span / self.time_granularity
784
785             customer_buckets = []
786             current_time = earliest_time
787
788             # Create evenly spaced buckets
789             for i in range(self.time_granularity):
790                 lower = current_time
791                 upper = min(current_time + bucket_size, latest_time)
792                 customer_buckets.append((lower, upper))
793                 current_time = upper
794
795                 if current_time >= latest_time:
796                     break
797
798             buckets[u] = customer_buckets
799
800         # Add single bucket for depot (start and end)
801         depot_earliest, depot_latest = self.time_windows[self.depot_start]
802         buckets[self.depot_start] = [(depot_earliest, depot_latest)]
803         buckets[self.depot_end] = [(depot_earliest, depot_latest)]
804
805         return buckets
806
807     def _create_time_graph(self):
808         """Create directed graph GT for time flow, updated to include
... service times"""
809         nodes_T = [] # (u, k, t-, t+)
810         edges_T = [] # ((u1,k1), (u2,k2))
811
812         # Create nodes for each customer and their time buckets
813         for u in self.customers:
814             for k, (t_min, t_max) in enumerate(self.T_u[u]):
815                 nodes_T.append((u, k, t_min, t_max))
816
817         # Add depot nodes
```

```
818         depot_start_bucket = self.T_u[self.depot_start][0]
819         depot_end_bucket = self.T_u[self.depot_end][0]
820         nodes_T.append((self.depot_start, 0, depot_start_bucket[0],
... depot_start_bucket[1]))
821         nodes_T.append((self.depot_end, 0, depot_end_bucket[0],
... depot_end_bucket[1]))
822
823         # Create edges between nodes, including service times in
... calculations
824         for i, k_i, t_min_i, t_max_i in nodes_T:
825             for j, k_j, t_min_j, t_max_j in nodes_T:
826                 if i != j:
827                     travel_time = self.costs[i,j] / 5
828                     service_time = self.service_times[i] # Service time at
... origin
829
830                     earliest_arrival = t_min_i + service_time + travel_time
831                     latest_arrival = t_max_i + service_time + travel_time
832
833                     if (earliest_arrival <= t_max_j and
834                         latest_arrival >= t_min_j and
835                         earliest_arrival <= self.time_windows[j][1] and
836                         latest_arrival >= self.time_windows[j][0]):
837                         edges_T.append(((i,k_i), (j,k_j)))
838
839         return nodes_T, edges_T
840
841     def _merge_capacity_buckets(self, dual_vars):
842         """Merge capacity buckets when their dual variables are equal"""
843         print("\nMerging capacity buckets...")
844         for u in self.customers:
845             print(f"\nCustomer {u}:")
846             print(f"    Before merge: {self.D_u[u]}")
847
848             buckets_to_merge = []
849
850             # Get all consecutive bucket pairs for this customer
851             for k in range(len(self.D_u[u]) - 1):
852                 i = (u, k) # First bucket node
853                 j = (u, k+1) # Next bucket node
854
855                 # Get dual variables for these nodes
856                 dual_i =
... dual_vars.get(f"capacity_flow_conservation_{u}_{k}", 0)
857                 dual_j =
... dual_vars.get(f"capacity_flow_conservation_{u}_{k+1}", 0)
858
859                 print(f"    Comparing buckets {k} and {k+1}:")
860                 print(f"    Dual values: {dual_i:.6f} vs {dual_j:.6f}")
```

```
861
862         # If duals are equal (within numerical tolerance), mark for
... merging
863         if abs(dual_i - dual_j) < 1e-6:
864             print(f"    -> Will merge")
865             buckets_to_merge.append((k, k+1))
866         else:
867             print(f"    -> Keep separate")
868
869         # Merge marked buckets (work backwards to avoid index issues)
870         for k1, k2 in reversed(buckets_to_merge):
871             lower = self.D_u[u][k1][0] # Lower bound of first bucket
872             upper = self.D_u[u][k2][1] # Upper bound of second bucket
873             # Remove the two original buckets and insert merged bucket
874             self.D_u[u].pop(k2)
875             self.D_u[u].pop(k1)
876             self.D_u[u].insert(k1, (lower, upper))
877
878         print(f"    After merge: {self.D_u[u]}")
879
880     def _merge_time_buckets(self, dual_vars):
881         """Merge time buckets when their dual variables are equal"""
882         print("\nMerging time buckets...")
883         for u in self.customers:
884             print(f"\nCustomer {u}:")
885             print(f"    Before merge: {self.T_u[u]}")
886
887             buckets_to_merge = []
888
889             # Get all consecutive bucket pairs for this customer
890             for k in range(len(self.T_u[u]) - 1):
891                 i = (u, k) # First bucket node
892                 j = (u, k+1) # Next bucket node
893
894                 # Get dual variables for these nodes
895                 dual_i = dual_vars.get(f"time_flow_conservation_{u}_{k}",
... 0)
896                 dual_j = dual_vars.get(f"time_flow_conservation_{u}_{k+1}",
... 0)
897
898                 print(f"    Comparing buckets {k} and {k+1}:")
899                 print(f"    Dual values: {dual_i:.6f} vs {dual_j:.6f}")
900
901                 # If duals are equal (within numerical tolerance), mark for
... merging
902                 if abs(dual_i - dual_j) < 1e-6:
903                     print(f"    -> Will merge")
904                     buckets_to_merge.append((k, k+1))
905                 else:
```

```
906         print(f"        -> Keep separate")
907
908     # Merge marked buckets (work backwards to avoid index issues)
909     for k1, k2 in reversed(buckets_to_merge):
910         lower = self.T_u[u][k1][0] # Lower bound of first bucket
911         upper = self.T_u[u][k2][1] # Upper bound of second bucket
912         # Remove the two original buckets and insert merged bucket
913         self.T_u[u].pop(k2)
914         self.T_u[u].pop(k1)
915         self.T_u[u].insert(k1, (lower, upper))
916
917     print(f"    After merge: {self.T_u[u]}")
918
919     def _is_significant_flow(self, flow, u_i, u_j):
920         """Determine if a flow is significant enough to trigger bucket
... expansion"""
921         # Flow should be significantly non-zero
922         if flow < 1e-4:
923             return False
924
925         # Skip flows between customers that are too far apart in time
926         # (these are less likely to be in optimal solution)
927         travel_time = self.costs[u_i, u_j] / 5
928         earliest_i = self.time_windows[u_i][0]
929         latest_j = self.time_windows[u_j][1]
930         if earliest_i + travel_time > latest_j - 10: # 10 time units
... buffer
931             return False
932
933         # Skip flows between customers that would exceed capacity
934         remaining_capacity = self.vehicle_capacity - self.demands[u_i]
935         if self.demands[u_j] > remaining_capacity * 0.9: # 90% threshold
936             return False
937
938         # Skip flows that would create very small buckets
939         min_bucket_size = (self.vehicle_capacity -
... min(self.demands.values())) * 0.1 # 10% of max remaining
940         if remaining_capacity < min_bucket_size:
941             return False
942
943         return True
944
945     def _expand_capacity_buckets(self, z_D):
946         """Add new capacity thresholds based on flow solution"""
947         print("\nExpanding capacity buckets...")
948
949         # First, create a mapping of flows to actual bucket indices
950         flow_mapping = {}
951         for (i, j), flow in z_D.items():
```

```
952         u_i, k_i = i # Source node customer and bucket index
953         u_j, k_j = j # Target node customer and bucket index
954
955         # Skip depot nodes and insignificant flows
956         if u_j in [self.depot_start, self.depot_end]:
957             continue
958
959         if not self._is_significant_flow(flow, u_i, u_j):
960             print(f"Skipping insignificant flow ({u_i},{k_i}) ->
... ({u_j},{k_j}) with flow {flow}")
961             continue
962
963         # Find current bucket indices after merging
964         source_bucket_idx = None
965         target_bucket_idx = None
966
967         # Find which bucket contains the original k_i index's value
968         for idx, (lower, upper) in enumerate(self.D_u[u_i]):
969             if k_i * ((self.vehicle_capacity - self.demands[u_i]) / 3)
... <= upper:
970                 source_bucket_idx = idx
971                 break
972
973         # Find which bucket contains the original k_j index's value
974         for idx, (lower, upper) in enumerate(self.D_u[u_j]):
975             if k_j * ((self.vehicle_capacity - self.demands[u_j]) / 3)
... <= upper:
976                 target_bucket_idx = idx
977                 break
978
979         if source_bucket_idx is not None and target_bucket_idx is not
... None:
980             flow_mapping[(u_i, source_bucket_idx, u_j,
... target_bucket_idx)] = flow
981
982         # Now process flows with correct bucket indices
983         for (u_i, k_i, u_j, k_j), flow in flow_mapping.items():
984             print(f"\nProcessing flow ({u_i},{k_i}) -> ({u_j},{k_j}) with
... flow {flow}")
985             print(f"Source customer {u_i} buckets: {self.D_u[u_i]}")
986             print(f"Target customer {u_j} buckets: {self.D_u[u_j]}")
987
988         # Calculate new threshold
989         d_plus_i = self.D_u[u_i][k_i][1] # Upper bound of source
... bucket
990         d_u_i = self.demands[u_i] # Demand of source customer
991         new_threshold = d_plus_i - d_u_i
992
993         print(f" New threshold calculated: {new_threshold}")
```

```
994         print(f"    Current buckets for customer {u_j}: {self.D_u[u_j]}")
995
996         # Add new threshold if it's not already present and it's
... meaningful
997         found = False
998         for bucket in self.D_u[u_j]:
999             if abs(bucket[1] - new_threshold) < 1e-6:
1000                 found = True
1001                 print("    -> Threshold already exists")
1002                 break
1003
1004         if not found and self.demands[u_j] < new_threshold <
... self.vehicle_capacity:
1005             print("    -> Adding new threshold")
1006             # Insert new bucket maintaining sorted order
1007             for k, bucket in enumerate(self.D_u[u_j]):
1008                 if new_threshold < bucket[1]:
1009                     # Split existing bucket at new threshold
1010                     if new_threshold > bucket[0]:
1011                         self.D_u[u_j].insert(k+1, (new_threshold,
... bucket[1]))
1012
1013                         self.D_u[u_j][k] = (bucket[0], new_threshold)
1014                         print(f"    Updated buckets: {self.D_u[u_j]}")
1015                         break
1016
1017 def _expand_time_buckets(self, z_T):
1018     """Add new time thresholds based on flow solution"""
1019     print("\nExpanding time buckets...")
1020
1021     # First, create a mapping of flows to actual bucket indices
1022     flow_mapping = {}
1023     for (i, j), flow in z_T.items():
1024         u_i, k_i = i # Source node customer and bucket index
1025         u_j, k_j = j # Target node customer and bucket index
1026
1027         # Skip depot nodes and insignificant flows
1028         if u_j in [self.depot_start, self.depot_end]:
1029             continue
1030
1031         if not self._is_significant_flow(flow, u_i, u_j):
1032             print(f"Skipping insignificant flow ({u_i},{k_i}) ->
... ({u_j},{k_j}) with flow {flow}")
1033             continue
1034
1035         # Find current bucket indices after merging
1036         source_bucket_idx = None
1037         target_bucket_idx = None
1038
1039         # Find which bucket contains the original k_i index's value
```



```
1039         time_span_i = self.time_windows[u_i][1] -
... self.time_windows[u_i][0]
1040         for idx, (lower, upper) in enumerate(self.T_u[u_i]):
1041             if k_i * (time_span_i / 3) <= upper:
1042                 source_bucket_idx = idx
1043                 break
1044
1045         # Find which bucket contains the original k_j index's value
1046         time_span_j = self.time_windows[u_j][1] -
... self.time_windows[u_j][0]
1047         for idx, (lower, upper) in enumerate(self.T_u[u_j]):
1048             if k_j * (time_span_j / 3) <= upper:
1049                 target_bucket_idx = idx
1050                 break
1051
1052         if source_bucket_idx is not None and target_bucket_idx is not
... None:
1053             flow_mapping[(u_i, source_bucket_idx, u_j,
... target_bucket_idx)] = flow
1054
1055         # Now process flows with correct bucket indices
1056         for (u_i, k_i, u_j, k_j), flow in flow_mapping.items():
1057             print(f"\nProcessing flow ({u_i},{k_i}) -> ({u_j},{k_j}) with
... flow {flow}")
1058             print(f"Source customer {u_i} buckets: {self.T_u[u_i]}")
1059             print(f"Target customer {u_j} buckets: {self.T_u[u_j]}")
1060
1061         # Calculate new threshold
1062         t_plus_i = self.T_u[u_i][k_i][1] # Upper bound of source
... bucket
1063         travel_time = self.costs[u_i,u_j] / 5 # Travel time between
... customers
1064         t_plus_j = self.T_u[u_j][k_j][1] # Upper bound of target
... bucket
1065
1066         new_threshold = min(t_plus_i - travel_time, t_plus_j)
1067
1068         print(f" New threshold calculated: {new_threshold}")
1069         print(f" Current buckets for customer {u_j}: {self.T_u[u_j]}")
1070
1071         # Add new threshold if it's not already present and it's
... meaningful
1072         found = False
1073         for bucket in self.T_u[u_j]:
1074             if abs(bucket[1] - new_threshold) < 1e-6:
1075                 found = True
1076                 print(" -> Threshold already exists")
1077                 break
1078
```



```
1079         if not found and self.time_windows[u_j][0] < new_threshold <
... self.time_windows[u_j][1]:
1080             print(" -> Adding new threshold")
1081             # Insert new bucket maintaining sorted order
1082             for k, bucket in enumerate(self.T_u[u_j]):
1083                 if new_threshold < bucket[1]:
1084                     # Split existing bucket at new threshold
1085                     if new_threshold > bucket[0]:
1086                         self.T_u[u_j].insert(k+1, (new_threshold,
... bucket[1]))
1087                         self.T_u[u_j][k] = (bucket[0], new_threshold)
1088                         print(f" Updated buckets: {self.T_u[u_j]}")
1089                         break
1090
1091     def _update_bucket_graphs(self):
1092         """Update time and capacity graphs after bucket modifications"""
1093         self.nodes_T, self.edges_T = self._create_time_graph()
1094         self.nodes_D, self.edges_D = self._create_capacity_graph()
1095
1096         print("\nUpdated graphs:")
1097         print(f"Time graph: {len(self.nodes_T)} nodes, {len(self.edges_T)}
... edges")
1098         print(f"Capacity graph: {len(self.nodes_D)} nodes,
... {len(self.edges_D)} edges")
1099
1100     def _reset_neighborhoods(self):
1101         """Reset LA neighborhoods to maximum size"""
1102         for u in self.customers:
1103             distances = [(j, self.costs[u,j]) for j in self.customers if j
... != u]
1104             distances.sort(key=lambda x: x[1])
1105             self.la_neighbors[u] = [j for j, _ in distances[:self.K]]
1106             print(f"Reset neighbors for customer {u}:
... {self.la_neighbors[u]}")
1107
1108     def _validate_buckets(self):
1109         """Validate bucket structures after modifications"""
1110         # Check capacity buckets
1111         for u in self.customers:
1112             # Verify bucket continuity
1113             prev_upper = None
1114             for i, (lower, upper) in enumerate(self.D_u[u]):
1115                 # Check bounds
1116                 if lower >= upper:
1117                     raise ValueError(f"Invalid capacity bucket bounds for
... customer {u}: [{lower}, {upper}]")
1118
1119                 # Check ordering
1120                 if prev_upper is not None and abs(lower - prev_upper) >
```

```
1120... 1e-6:
1121         raise ValueError(f"Gap in capacity buckets for customer
... {u} between {prev_upper} and {lower}")
1122
1123         # Check within vehicle capacity
1124         if upper > self.vehicle_capacity:
1125             raise ValueError(f"Capacity bucket for customer {u}
... exceeds vehicle capacity: {upper}")
1126
1127         prev_upper = upper
1128
1129         # Check time buckets
1130         for u in self.customers:
1131             # Verify bucket continuity
1132             prev_upper = None
1133             for i, (lower, upper) in enumerate(self.T_u[u]):
1134                 # Check bounds
1135                 if lower >= upper:
1136                     raise ValueError(f"Invalid time bucket bounds for
... customer {u}: [{lower}, {upper}]")
1137
1138                 # Check ordering
1139                 if prev_upper is not None and abs(lower - prev_upper) >
... 1e-6:
1140                     raise ValueError(f"Gap in time buckets for customer {u}
... between {prev_upper} and {lower}")
1141
1142                 # Check within time windows
1143                 if upper > self.time_windows[u][1] or lower <
... self.time_windows[u][0]:
1144                     raise ValueError(f"Time bucket for customer {u} outside
... time window: [{lower}, {upper}]")
1145
1146                 prev_upper = upper
1147
1148 def load_solomon_instance(filename, customer_ids=None):
1149     """
1150     Load Solomon VRPTW instance from CSV file
1151
1152     Args:
1153         filename: Path to CSV file
1154         customer_ids: List of specific customer IDs to include (None for
... all)
1155     """
1156     # Read CSV file
1157     df = pd.read_csv(filename)
1158
1159     # Convert 'Depot' to 0 in CUST_NUM
1160     df['CUST_NUM'] = df['CUST_NUM'].replace('Depot', '0')
```

```
1161     df['CUST_NUM'] = df['CUST_NUM'].astype(int)
1162
1163     # Filter customers if specific IDs provided
1164     if customer_ids is not None:
1165         selected_ids = [0] + sorted(customer_ids)
1166         df = df[df['CUST_NUM'].isin(selected_ids)]
1167
1168         if len(df) != len(selected_ids):
1169             missing = set(selected_ids) - set(df['CUST_NUM'])
1170             raise ValueError(f"Customer IDs not found: {missing}")
1171
1172     # Extract coordinates
1173     coords = {row['CUST_NUM']: (row['XCOORD.'], row['YCOORD.'])
1174               for _, row in df.iterrows()}
1175
1176     # Create customer list (excluding depot)
1177     customers = sorted(list(set(df['CUST_NUM']) - {0}))
1178
1179     # Add virtual end depot with same coordinates as start depot
1180     virtual_end = max(customers) + 1
1181     coords[virtual_end] = coords[0]
1182
1183     # Calculate costs/distances (not including service times)
1184     costs = {}
1185     all_nodes = [0] + customers + [virtual_end]
1186     for i in all_nodes:
1187         for j in all_nodes:
1188             if i != j:
1189                 x1, y1 = coords[i]
1190                 x2, y2 = coords[j]
1191                 costs[i,j] = np.floor(np.sqrt((x2-x1)**2 + (y2-y1)**2) *
... 10) / 10
1192
1193     # Extract service times (0 for depots)
1194     service_times = {row['CUST_NUM']: row['SERVICE_TIME']
1195                     for _, row in df.iterrows()}
1196     service_times[virtual_end] = 0 # No service time at end depot
1197
1198     # Extract time windows and demands
1199     time_windows = {row['CUST_NUM']: (row['READY_TIME'], row['DUE_DATE'])
1200                    for _, row in df.iterrows()}
1201     time_windows[virtual_end] = time_windows[0] # End depot has same time
... window as start
1202
1203     demands = {row['CUST_NUM']: row['DEMAND']
1204               for _, row in df.iterrows()}
1205     demands[virtual_end] = 0
1206
1207     # Create instance dictionary
```

```
1208     instance = {
1209         'customers': customers,
1210         'depot_start': 0,
1211         'depot_end': virtual_end,
1212         'costs': costs, # Just distances between customers
1213         'time_windows': time_windows,
1214         'service_times': service_times, # Added service times
1215         'demands': demands,
1216         'vehicle_capacity': 200
1217     }
1218
1219     print("\nSelected Customer Details:")
1220     print("ID Ready Due Service Demand Location")
1221     print("-" * 50)
1222     for c in customers:
1223         x, y = coords[c]
1224         tw = time_windows[c]
1225         print(f"{c:<3} {tw[0]:<6} {tw[1]:<6} {service_times[c]:<8}
... {demands[c]:<7} ({x},{y})")
1226
1227     return instance
1228
1229 def run_solomon_instance(filename, customer_ids, K=3, time_granularity=3,
... capacity_granularity=3, max_iterations=5, time_limit=300):
1230     print(f"Loading Solomon instance with {len(customer_ids)} selected
... customers...")
1231     instance = load_solomon_instance(filename, customer_ids)
1232
1233     optimizer = VRPTWOptimizer(
1234         customers=instance['customers'],
1235         depot_start=instance['depot_start'],
1236         depot_end=instance['depot_end'],
1237         costs=instance['costs'],
1238         time_windows=instance['time_windows'],
1239         service_times=instance['service_times'], # Added service times
1240         demands=instance['demands'],
1241         vehicle_capacity=instance['vehicle_capacity'],
1242         K=K,
1243         time_granularity=time_granularity,
1244         capacity_granularity=capacity_granularity,
1245         max_iterations=max_iterations
1246     )
1247
1248     print("\nSolving...")
1249     solution = optimizer.solve_with_parsimony(time_limit=time_limit)
1250
1251     return optimizer, solution
1252
1253 def test_phi_functions():
```

```
1254     """Test the LA Arc computation with multiple test cases"""
1255     # Test case data
1256     time_windows = {
1257         1: (10, 40),      # Customer 1
1258         2: (30, 70),      # Customer 2
1259         3: (50, 80),      # Customer 3
1260         4: (20, 60),      # Customer 4
1261         5: (40, 90),      # Customer 5
1262     }
1263
1264     # Customer locations
1265     locations = {
1266         1: (2, 4),        # Customer 1
1267         2: (-1, 3),       # Customer 2
1268         3: (4, 1),        # Customer 3
1269         4: (-2, -3),      # Customer 4
1270         5: (1, -2),       # Customer 5
1271     }
1272
1273     # Calculate all travel times
1274     travel_times = {}
1275     for i in locations:
1276         for j in locations:
1277             if i != j:
1278                 x1, y1 = locations[i]
1279                 x2, y2 = locations[j]
1280                 dist = np.floor(np.sqrt((x2-x1)**2 + (y2-y1)**2) * 10) / 10
1281                 travel_times[(i,j)] = dist
1282
1283     la_computer = LAArcComputer(time_windows, travel_times)
1284
1285     # Test Case 1: Simple two-customer sequence
1286     print("\nTest Case 1: Two customers [1,2]")
1287     r1 = [1, 2]
1288     phi_r1 = la_computer.compute_phi_r(r1)
1289     phi_hat_r1 = la_computer.compute_phi_hat_r(r1)
1290     print(f"φ_r: {phi_r1}, φ̂_r: {phi_hat_r1}")
1291     # The earliest departure should be earlier than the latest departure
1292     assert phi_r1 <= phi_hat_r1, "Earliest departure should be before
... latest departure"
1293     # Both values should be within the first customer's time window
1294     assert time_windows[1][0] <= phi_r1 <= time_windows[1][1], "φ_r outside
... time window"
1295     assert time_windows[1][0] <= phi_hat_r1 <= time_windows[1][1], "φ̂_r
... outside time window"
1296
1297     # Test Case 2: Three-customer sequence
1298     print("\nTest Case 2: Three customers [1,2,3]")
1299     r2 = [1, 2, 3]
```

```
1300     phi_r2 = la_computer.compute_phi_r(r2)
1301     phi_hat_r2 = la_computer.compute_phi_hat_r(r2)
1302     print(f"φ_r: {phi_r2}, φ̂_r: {phi_hat_r2}")
1303     assert phi_r2 <= phi_hat_r2
1304
1305     # Test Case 3: Different three-customer sequence
1306     print("\nTest Case 3: Three customers [4,5,1]")
1307     r3 = [4, 5, 1]
1308     phi_r3 = la_computer.compute_phi_r(r3)
1309     phi_hat_r3 = la_computer.compute_phi_hat_r(r3)
1310     print(f"φ_r: {phi_r3}, φ̂_r: {phi_hat_r3}")
1311     assert phi_r3 <= phi_hat_r3
1312
1313     # Detailed analysis of results
1314     print("\nDetailed Analysis:")
1315     print("Travel times between customers:")
1316     for (i,j), time in sorted(travel_times.items()):
1317         print(f"From {i} to {j}: {time}")
1318
1319     print("\nTime windows:")
1320     for i, window in sorted(time_windows.items()):
1321         print(f"Customer {i}: {window}")
1322
1323     '''if __name__ == "__main__":
1324         print("Testing LA Arc computation...")
1325         test_phi_functions()'''
1326
1327 if __name__ == "__main__":
1328     optimizer, solution = run_solomon_instance(
1329         filename="r109.csv",
1330         # 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],
1331         customer_ids=list(range(1,51)),
1332         K=10,
1333         time_granularity=10,
1334         capacity_granularity=10,
1335         max_iterations=5,
1336         time_limit=300
1337     )
```