Machine Learning based approach

Dataset:
Orders:

- Total: 12 orders
- Capacity demand: ~32.14 units
- Time windows: Average 180 min width
- Types: 8 TypeA (0.75 cap), 4 TypeB (1.5 cap)

Fleet:

- 10 vehicles total
- Types: Van (2.5 cap), TruckSmall (6.0), TruckLarge (12.0), Bike (0.5), MiniTruck (4.0)
- Total fleet capacity: ~44 units (sufficient)

Constraints:

- Multiple vehicle-location restrictions
- Break required every ~5.5 hours
- Only certain nodes allow breaks

Challenges:

- Node 2: Popular delivery (5 orders)
- Node 11: Popular pickup (4 orders)
- Tight time windows require sequential routing
- Vehicle restrictions limit flexibility

Key 👍

Pickup-Delivery Problem (PDP)

- Each order has TWO locations (pickup + delivery)
- Must pickup before delivery (precedence)
- Same vehicle handles both

Time Windows

- Each order has [earliest, latest] pickup time
- Service must arrive within window
- Waiting is allowed (creates idle time)

Capacity Tracking

- Load increases at pickup
- Load decreases at delivery
- Must check capacity at EVERY point in route

Service Constraints

- Max 456 min (7.6 hours) including break
- Must start and end at depot
- One 30-min break before 4.5h driving

Phase 1: Greedy Constructor

1. Construction of Greedy Solution

SimpleGreedyConstructor - Builds initial solution

Algorithm:

1. Sort orders by earliest time window
2. For each vehicle:
   - Create new service
   - Greedily insert compatible orders
   - Add break
3. Return solution with assigned orders

ConstraintValidator - Checks solution feasibility

Validates:

- ✅ A. Service Constraints
  - Service returns to origin

    - Max duration ≤ 456 min (7.6 hours)
- ✅ B. Time Windows
    - Pickup within [from_time, to_time]
- ✅ C. Vehicle Constraints
    - Capacity not exceeded at any point
    - Vehicle can visit all locations
    - Precedence: pickup before delivery
- ✅ D. Break Constraints
    - Exactly one 30-min break
    - Before 270 min driving or 330 min working
    - At break-allowed location

Cost:

Component 4: Objective Function (`constraint_validator.py`)

ObjectiveCalculator - Computes solution quality

```
objective = ObjectiveCalculator(problem, weights={
    'num_services': 100.0,
    'num_vehicles': 150.0,
    'total_distance': 1.0,
    'total_idle_time': 0.5,
    'unserved_penalty': 1000.0
})

total_cost, components = objective.calculate(solution)
```

Objective Components:

- Minimize: Number of services (routes)
- Minimize: Number of vehicles used
- Minimize: Total distance traveled
- Minimize: Total idle time
- Maximize: Order coverage (minimize unserved)

```
┌──────────────────────────────────────────────────────────────┐
│                  PHASE 0: FOUNDATION                         │
├──────────────────────────────────────────────────────────────┤
│                                                              │
│  ┌──────────────┐     ┌──────────────────┐                   │
│  │  CSV Files   │────▶│ ProblemInstance  │                   │
│  └──────────────┘     └────────┬─────────┘                   │
│                                │                             │
│                                ▼                             │
│              ┌─────────────────────────┐                     │
│              │ SimpleGreedy            │                     │
│              │ Constructor             │                     │
│              └────────────┬────────────┘                     │
│                           │                                  │
│                           ▼                                  │
│                ┌──────────────┐                              │
│                │  Solution    │                              │
│                └──────┬───────┘                              │
│                       │                                      │
│        ┌──────────────┴──────────────────┐                   │
│        ▼                                 ▼                   │
│  ┌─────────────────────┐      ┌──────────────────┐           │
│  │ Constraint          │      │ Objective        │           │
│  │ Validator           │      │ Calculator       │           │
│  └─────────────────────┘      └──────────────────┘           │
│        │                                 │                   │
│        ▼                                 ▼                   │
│   Feasibility?                     Total Cost                │
│   Violations                       Metrics                   │
│                                                              │
└──────────────────────────────────────────────────────────────┘
```

Phase 2: ALNS Operator

Algorithm:

```None
1. Start with greedy solution (from Phase 0)
2. Loop for N iterations:
   a. SELECT destroy operator (weighted random)
```

```
   b. SELECT repair operator (weighted random)
   c. DESTROY: Remove 1-3 orders from solution
   d. REPAIR: Reinsert orders optimally
   e. ACCEPT/REJECT using Simulated Annealing
   f. UPDATE operator weights based on performance
3. Return best solution found
```

Destroy Operators:

- RandomRemoval: Randomly remove orders
- WorstRemoval: Remove orders with highest cost
- ShawRemoval: Remove related orders (similar in time/space)

Repair Operators:

- GreedyInsertion: Insert at lowest cost position
- Regret2Insertion: Prioritize hard-to-insert orders

1. Adaptive Weights

Operators that find better solutions get higher weights → used more often

2. Simulated Annealing

Accepts worse solutions early (exploration) → only better solutions later (exploitation)

3. Destroy-Repair

Break solution apart → rebuild better → escape local optima

4. Operator Portfolio

Multiple operators work on different problem aspects:

- Random: Diversification
- Worst: Remove expensive routes
- Shaw: Exploit spatial/temporal clustering

Phase/part 3: Training ALNS on RL

perator selection in ALNS, using DQN (Deep Q-Network) with experience replay. The implementation maintains 91.7% order coverage while achieving modest cost improvements (0.36%). The framework provides a solid foundation for more sophisticated architectures.

System components:

```
┌──────────────────────────────────────────────┐
│          Phase 2a: RL-based ALNS             │
├──────────────────────────────────────────────┤
│                                              │
│  ┌─────────────┐         ┌─────────────┐     │
│  │ Solution    │──encode──►│ State     │     │
│  │ (Current)   │         │ Encoder     │     │
│  └─────────────┘         └──────┬──────┘     │
│                   │ 19D vector               │
│                   ▼                          │
│            ┌─────────────┐                   │
│            │ DQN Network │                   │
│            │ (4 layers)  │                   │
│            └──────┬──────┘                   │
│                   │ Q-values                 │
│                   ▼                          │
│           ┌──────────────┐                   │
│           │ Operator Pair│                   │
│           │  Selection   │                   │
│           └──────┬───────┘                   │
│                  │                           │
│       ┌──────────┴──────────┐                │
│       ▼                     ▼                │
│  ┌──────────┐          ┌──────────┐          │
│  │ Destroy  │          │ Repair   │          │
│  │ Operator │          │ Operator │          │
```

```
│          └──────────────────────┘          └──────────────────────┘ │
│             │                                                        
│              └──────────────────────────────────────────────┘   │
│                      ▼                                              
│                  ┌──────────────────────┐                          │
│                  │ New Solution │                                   
│                  └──────────────────────┘                          │
│                      │                                              
│                      ▼                                              
│                  ┌──────────────────────┐                          │
│                  │  Reward  │                                       
│                  │ Calculation │                                    
│                  └──────────────────────┘                          │
│                      │                                              
│                      ▼                                              
│                  ┌──────────────────────┐                          │
│                  │  Update  │                                       
│                  │ Q-Network │                                      
│                  └──────────────────────┘                          │
│                                                                     
└──────────────────────────────────────────────────────────────────────
─────┘
```

. Technical Implementation

2.1 State Representation

The solution state is encoded into a 19-dimensional feature vector:

```
State Features (normalized to [0, 1]):
1.  num_services / 10.0
2.  num_vehicles_used / 10.0
3.  unserved_orders / total_orders
4.  coverage_rate
5.  total_distance / 1000.0
6.  total_driving_time / 1000.0
7.  total_idle_time / 500.0
8.  num_violations / 20.0
9.  duration_violations / 10.0
10. capacity_violations / 10.0
```

```
11. time_window_violations / 10.0
12. break_violations / 10.0
13. precedence_violations / 10.0
14. other_violations / 10.0
15. mean_capacity_utilization
16. max_capacity_utilization
17. min_capacity_utilization
18. mean_duration_utilization
19. max_duration_utilization
```

Design rationale: Simple feedforward features that capture solution quality without requiring graph encoding. Computationally efficient, suitable for real-time optimization.

2.2 Action Space

Actions: All (destroy, repair) operator pairs

- 3 destroy operators × 2 repair operators = 6 actions

Operators:

- Destroy: RandomRemoval, WorstRemoval, ShawRemoval
- Repair: GreedyInsertion, Regret2Insertion

Action selection: Epsilon-greedy with decay

- Initial epsilon: 1.0 (full exploration)
- Final epsilon: 0.1 (10% exploration)
- Decay rate: 0.995 per episode

2.3 DQN Architecture

```
Network Structure:
Input Layer:    19 features
Hidden Layer 1: 128 neurons + ReLU
Hidden Layer 2: 64 neurons + ReLU
Hidden Layer 3: 32 neurons + ReLU
Output Layer:    6 Q-values (one per action)

Optimizer: Adam (lr=0.001)
Loss: MSE between predicted and target Q-values
```

Key techniques:

- Experience replay (buffer size: 2000)
- Target network (updated every 5 episodes)
- Batch training (32 samples)
- Gamma: 0.95 (discount factor)

2.4 Reward Function

```
Reward Components:
1. Violation reduction: +50 per violation removed
2. Coverage improvement: +100 per % coverage gained
3. Cost improvement:
   - New global best: +100 + improvement%×100
   - Better than current: +10 + improvement%×50
   - Within 5% worse: +1
   - Much worse: -5

Priority: Feasibility > Coverage > Cost
```

Design decisions:

- Feasibility heavily rewarded (addresses constraint violations)
- Coverage protected (prevents dropping orders)
- Cost optimization secondary (only after feasibility maintained)

2.5 Training Configuration

```
Default Parameters:
- Episodes: 20
- Steps per episode: 100
- Total iterations: 2000
- Destroy size: 1-3 orders
- Update frequency: Every 5 episodes
- Batch size: 32
```

---

3. Experimental Results

3.1 Dataset Characteristics

Problem Instance:

- Orders: 12 (8 TypeA, 4 TypeB)
- Vehicles: 10 (5 types)
- Total capacity demand: 32.14 units
- Fleet capacity: 40.50 units
- Time window tightness: Average 180 min

Initial Solution (Greedy):

- Services: 6
- Coverage: 91.7% (11/12 orders)
- Violations: 5 (all duration-related)
- Cost: 27,092.94
- Unserved: Order 2 (7.06 capacity - too large)
-

None

```
Configuration: unserved_penalty = 10,000 (10x increase)
Episodes: 20
Best Episode: #3
Cost: 26,995.16 (0.36% improvement)
Coverage: 91.7% (maintained)
Violations: 5 (unchanged)
Verdict: SUCCESS - Coverage protected
```

3.3 Performance Analysis

Coverage Stability:

- Minimum: 91.7%
- Maximum: 91.7%
- Average: 91.7%
- Std Dev: 0.0%

Result: Perfect stability across all 20 episodes.

Cost Trajectory:

- Episode 1: 27,030.77
- Episode 3: 26,995.16 (best)
- Episode 20: 27,092.94
- Improvement: 0.36%

Violation Analysis:

- Initial: 5 violations (all SERVICE_DURATION)
- Final: 5 violations (unchanged)
- Reason: Structural problem - routes inherently too long

Learning Evidence:

- Epsilon decay: 1.0 → 0.1 (proper exploration/exploitation)
- Reward trend: Negative early → Positive later
- Operator preferences: Regret2Insertion learned (54% usage)
- Q-value convergence: Stable after episode 10

---

4. Key Findings

4.1 What Works

1. RL Infrastructure: Solid implementation
    - DQN training stable
    - Experience replay effective
    - Target network prevents divergence
2. Coverage Protection: Perfect
    - 91.7% maintained throughout
    - Objective function balance correct
    - Agent learns to preserve orders
3. Operator Selection: Learning detected
    - Regret2Insertion preferred (54% vs 46%)
    - WorstRemoval less used (26% vs 45%)
    - Policy converges after ~500 iterations

4.2 Limitations

1. Minimal Improvement: Only 0.36%

    - Problem is structurally difficult
    - Cannot create new services
    - Duration violations unfixable with shuffling
2. Constraint Violations: Unchanged
    - 5 violations persist
    - All SERVICE_DURATION type
    - Routes 130-178 minutes over limit
3. State Representation: Too simple
    - 19 features insufficient for complex decisions
    - No graph structure captured
    - Missing temporal patterns
4. Action Space: Limited
    - Only 6 operator pairs
    - No hyperparameter control
    - Cannot adjust destroy size

4.3 Lessons Learned

Objective Function Balance:

- Critical to tune penalty weights
- Coverage loss easy if weights wrong
- Feasibility > Coverage > Cost priority essential

Training Requirements:

- 250 iterations insufficient
- 2000 iterations adequate for convergence
- Episode structure works well

Problem Characteristics:

- Initial solution quality limits improvement
- Structural constraints need different operators
- May need service creation capability

---

5. Comparison: Classical ALNS vs RL-based ALNS

Aspect	Phase 1 (Classical)	Phase 2a (RL-based)
Operator Selection	Adaptive weights (statistical)	Q-network (learned)
Exploration	Fixed noise factor	Epsilon-greedy decay
Memory	None	Experience replay (2000)
Adaptation Speed	Immediate	Gradual (batch updates)
Iterations	500	2000
Coverage	83.3%	91.7%
Violations	5	5
Cost Reduction	0.74%	0.36%
Computational Cost	Low	Medium

Verdict: RL adds complexity but improves stability. For this specific problem, classical ALNS comparable due to structural constraints.

DQN Parameters:
- learning_rate: 0.001
- gamma: 0.95
- epsilon_start: 1.0
- epsilon_end: 0.1
- epsilon_decay: 0.995
- batch_size: 32
- memory_size: 2000

Training Parameters:
- num_episodes: 20
- steps_per_episode: 100
- destroy_size_min: 1
- destroy_size_max: 3
- update_frequency: 5

Objective Weights:
- num_services: 100.0
- num_vehicles: 150.0
- total_distance: 1.0
- total_idle_time: 0.5
- unserved_penalty: 10,000.0
- violation_penalty: 5,000.0 per violation