DDCM GPU Implementation

Technical Deep Dive & Code Walkthrough

December 2024 Update: Universal Graph + BI Optimization

9.3× Additional BI Speedup via Basis Function Approximation

Press Space for next page →

Technical Foundation

Core Concepts & Entry Points

Entry Point: Universal Batch Simulation

File: `calibration/batch_simulate_universal.py` (NEW - Dec 2024)

The main driver orchestrates simulation for thousands of agents.

KEY RESPONSIBILITIES:

- **Population Loading**: Load agent profiles (Home, Work, Sequence).
- **Grouping by SEQUENCE**: Groups by `(seq_key, has_child)` NOT home zone.
- **Universal Graph**: Build ONE graph for ALL home zones.
- **Per-Home BI**: Run backward induction per home zone.
- **Parallel Simulation**: Generate schedules using GPU.

Old vs New Grouping Strategy

OLD APPROACH (`BATCH_SIMULATE.PY`)

```
# Groups by: home_zone, mandatory_sequence, has_child
# Result: 144 zones × N sequences = MANY groups
grouped = df.groupby(['home_zone_str', 'seq_key', 'has_child'])
# Each group builds separate 60s graph → 144 × 60s = 8,640s
```

NEW APPROACH (`BATCH_SIMULATE_UNIVERSAL.PY`)

```
# Groups by: mandatory_sequence, has_child ONLY
# Result: N sequences = FEW groups
grouped = df.groupby(['seq_key', 'has_child'])
# ONE graph (60s) + 144 × 2s BI = 357s → 24.7× faster!
```

Batch Simulation Code Flow

```python
# calibration/batch_simulate_universal.py

# 1. Build Universal Graph (ONE for all home zones)
states, graph_data, home_indices, _ = run_universal_forward_pass(
    od_lookup, zone_attr, initial_states,  # Multiple homes!
    mandatory_sequence, device='cuda'
)

# 2. Construct CSR Graph (ONCE)
graph = builder.build_graph(states, graph_data)

# 3. Backward Induction (PER HOME ZONE)
for home_zone in unique_home_zones:
    V_tensors[home_zone] = solver.run(
        states, graph, home_zone_id=home_zone.value - 1
    )

# 4. Simulate Agents (per home, shared graph)
plans = simulate_batch_tensor(...)
```

Core Technology: PyTorch & CUDA

THE PROBLEM

- Python loops are too slow for 10^6 states
- Object overhead kills memory (millions of `State` objects)

THE SOLUTION

- **Tensors**: Represent states as GPU matrices `(N, 8)`
- **Parallelism**: Operations for 100,000+ states **simultaneously**
- **CSR Format**: Sparse graph with O(1) edge lookup

Core Technology: Level-Synchronous BFS

DEFINITION

A Breadth-First Search that processes the graph layer-by-layer.

IN DDCM CONTEXT

- **Layers = Time Steps**: Process t=9:00, then t=9:15, then t=9:30
- **Synchronization**: Finish ALL states at t before moving to t+15
- **Deduplication**: Merge identical states at each time step

WHY IT MATTERS

- Maximizes GPU parallelism (SIMD execution)
- Enables efficient global deduplication via `torch.unique`
- Prevents exponential state explosion

State Representation

STATE TENSOR: (N, 8) INTEGER TENSOR

Column	Name	Description	Range
0	Time	Minutes from midnight	0-1440
1	Zone	Current location	1-144
2	Activity	Current activity type	0-9
3	Duration	Time spent in activity	0-31
4	Mode	Transport mode used	0-7
5	Car In Use	Is car elsewhere	0/1
6	Moto In Use	Is motorcycle elsewhere	0/1

Theoretical Foundations

Bridging Engineering to Academic Nomenclature

Terminology Map

Your Term (Engineering)	Academic Synonym (Control/Formal)	Academic Synonym (AI/Planning)
Condition (Hard)	State Invariant / Safety Constraint	Pruning Rule / Hard Constraint
Condition (Soft)	Viability Cost / Soft Barrier	Preference / Soft Constraint
Forward Reachability	Forward Reachable Set (FRS)	Reachability Graph
Forward Reachable Tube	Flowpipe / Reachable Tube	Trajectory Envelope
Condition-Based Pruning	Safety Filter / Infeasibility Pruning	Constraint Propagation

Implementation Bridges

Your Implementation	Theoretical Concept	Why it fits
Universal Graph	**State Aggregation / Bisimulation**	Merging states with identical future dynamics.
Basis Function BI	**Linear Value Function Approx (VFA)**	Approximating V as linear combination of features.
Multi-Fidelity BI	**Surrogate Modeling**	Using high-cost "truth" to train low-cost surrogate.
Level-Sync BFS	**Symbolic Reachability**	Processing sets of states as tensors (symbolic).

Basis Function Approximation (Powell)

FROM VALUE FUNCTIONS TO VFAS

- **Value Function** $V_t(S_t)$: Expected cumulative utility from state S_t.
- **Problem**: State space explosion (Curse of Dimensionality).
- **Solution**: Approximate $V_t(S_t)$ with $\hat{V}_t(S_t;\theta)$ parameterized by θ.

WHAT ARE BASIS FUNCTIONS?

Features $f_k(S_t)$ used to construct the approximator:

$$\hat{V}(S) = \sum_k \theta_k f_k(S)$$

- **Examples**: Linear terms, quadratic/interaction terms, non-linear transformations.

Why Basis Functions are Needed

1. CURSE OF DIMENSIONALITY

Avoids the need for value estimates for an astronomically large grid of states.

2. GENERALIZATION

Allows nearby states to share information. Learned parameters θ imply values for unvisited states.

3. STRUCTURE EXPLOITATION

Encodes known properties (linearity, concavity, monotonicity) to get faster convergence.

4. LEARNING WITH LIMITED DATA

Low-dimensional θ can be estimated reliably from relatively little simulation data.

How Basis Functions enter DDCM (1/2)

1. STATE AND DECISION

S_t (activity, time, durations) and a_t (chosen alternative).

2. APPROXIMATE CONTINUATION VALUE

Next-state S_{t+1} depends on S_t, a_t. Approximate value:

$$\hat{V}(S_{t+1}; \theta) = \sum_k \theta_k f_k(S_{t+1})$$

How Basis Functions enter DDCM (2/2)

3. POLICY DEFINED BY VFA

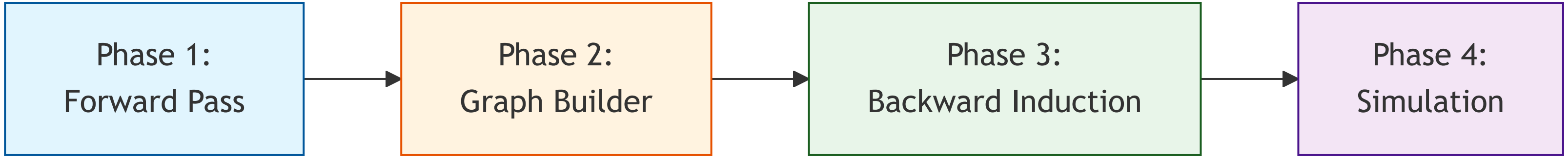
Choose a_t by maximizing current plus approximate future value:

$$\pi(S_t) = \arg\max_a \{C(S_t, a) + \gamma \mathbb{E}[\hat{V}(S_{t+1}; \theta) \mid S_t, a]\}$$

Implementation Walkthrough

From Code to Execution

Overall Architecture



Phase	Time	Optimized (Basis Func.)
Forward Pass	~30s	~30s
Graph Build	~29s	~29s
Backward Induction	~2s/home × 144 = 288s	**20s total (9.3×)**
Simulation	~0.4s	~0.4s

Phase 1: Forward Pass

Discovering the State Space

Phase 1: Forward Pass Overview

Goal: Discover all reachable states from initial state(s).

File: `planning/forward_pass_tensor.py`

TWO FUNCTIONS AVAILABLE:

Function	Use Case	Input
`run_tensor_forward_pass()`	Single home zone	1 initial state
`run_universal_forward_pass()`	Multiple home zones	List of initial states

OUTPUT:

- `states_tensor`: (N, 8) tensor of unique states
- `graph_data`: Edge information for GraphBuilder
- `home_state_indices`: Mapping home → initial state index

Forward Pass: Single Origin

Function: `run_tensor_forward_pass()`

```
states_tensor, graph_data, stats = run_tensor_forward_pass(
    od_lookup,           # ODLookupOptimized object
    zone_attr,           # Zone attributes DataFrame
    initial_state,       # State(time=0, zone=HOME, ...)
    has_child=False,     # Agent attribute
    mandatory_sequence=[(WORK, Zone.CZONE_100)],
    delta_t=15,          # 15-minute time steps
    end_time=1440,       # End of day
    device='cuda',       # GPU acceleration
    return_tensors=True, # Return graph_data
    reachability_masks=masks  # Pre-computed reachability
)
```

Performance: ~30s for 1.5M states, 326M edges

Forward Pass: Multi-Origin (Universal)

Function: `run_universal_forward_pass()` (NEW - Dec 2024)

```python
# Create initial states for ALL home zones
initial_states = [
    State(time=0, zone=home, activity=HOME, ...)
    for home in [Zone.CZONE_1, Zone.CZONE_2, ..., Zone.CZONE_144]
]

# Single forward pass for 144 homes!
states, graph_data, home_indices, stats = run_universal_forward_pass(
    od_lookup, zone_attr, initial_states,
    has_child=False,
    mandatory_sequence=[(WORK, Zone.CZONE_100)],
    device='cuda',
    reachability_masks=masks
)
```

Key Insight: States stay ~1.53M for 1 home OR 144 homes!

Why State Count Stays Constant

THE MAGIC OF DEDUPLICATION

After t=300 (5:00 AM), paths from different homes **converge**:

```
t=0:   144 different starting states (one per home)
t=300: 4,503 states (some overlap)
t=600: 12,236 states (heavy overlap)
t=840: 24,454 states (almost identical to 1-home!)
```

WHY?

Once you leave home, your state is defined by:

- Current zone (not where you started)
- Current activity, mode, time
- Mandatory progress

Your home zone doesn't affect the graph structure!

Forward Pass: Internal Implementation

Class: `TensorForwardPass`
Method: `run_multi_origin()`

TIME-BUCKET BFS ALGORITHM (1/2):

```
for t in range(0, end_time + 1, delta_t):
    # 1. GATHER: Collect states scheduled for time t
    tensors_at_t = pending_states.pop(t, [])
    current_batch = torch.cat(tensors_at_t, dim=0)

    # 2. GPU TRANSFER (Fixed Dec 2024!)
    current_batch = current_batch.to(device)

    # 3. DEDUPLICATE: Merge identical states
    unique_states = torch.unique(current_batch, dim=0)
```

Forward Pass: Internal Implementation (2/2)

TIME-BUCKET BFS ALGORITHM (2/2):

```
# 4. PRUNE: Remove unreachable states (Fixed Dec 2024!)
unique_states = prune_unreachable_states(unique_states, t)

# 5. EXPAND: Generate next states via GPU kernel
next_states = kernel.expand_states_batch_tensor(unique_states)

# 6. SCATTER: Schedule next states to future time buckets
for next_t in unique(next_states[:, 0]):
    pending_states[next_t].append(next_states[next_states[:, 0] == next_t])
```

GPU Kernel: 3-Stage Pipeline

File: `planning/gpu_kernels.py`

STAGE 1: ACTION GENERATION (`GPUACTIONGENERATOR`)

- Generate ALL possible actions via broadcasting
- `torch.meshgrid(zones, modes)` → 144 × 8 = 1,152 candidates/state

STAGE 2: CONSTRAINT FILTERING (`GPUCONSTRAINTFILTER`)

- Pre-computed boolean masks: `(96 time steps, 10 activities, 8 modes)`
- Vectorized lookup: `valid = mask[t, activity, mode]`
- **~49% actions filtered** with ReachabilityMasks

STAGE 3: TRANSITION COMPUTATION

- Compute next states and utilities
- OD travel time lookup (vectorized)

Constraint Filtering Example

Question: "Can I start SHOPPING at 8:00 AM?"

PRE-COMPUTED MASK (GPU TENSOR)

```
        HOME  WORK  SHOP  LEISURE ...
 8:00    ✓     ✓     ✗      ✗
 9:00    ✓     ✓     ✓      ✗
10:00    ✓     ✓     ✓      ✓
  ...
```

VECTORIZED LOOKUP (NO PYTHON LOOP!)

```
# For 100,000 actions at once:
times = actions[:, 0] // 15  # Convert to time index
activities = actions[:, 1]
valid = constraint_mask[times, activities]  # Boolean tensor
filtered_actions = actions[valid]
```

ReachabilityMasks: Geographic Filtering

File: `model/zone_prefilter.py`

PRE-COMPUTED FOR EACH (ORIGIN, MODE):

```
# Can I reach zone J from zone I using mode M within time budget?
reachable[I, J, M] = travel_time[I, J, M] < max_budget[M]
```

MODE-SPECIFIC COVERAGE (144 ZONES):

Mode	Avg Reachable	Coverage
CAR	143.0	99.3%
BUS	116.3	80.8%
BICYCLE	38.4	26.6%
WALK	2.7	1.9%

Phase 2: Graph Construction

Building the CSR Sparse Graph

Phase 2: Graph Builder Overview

Goal: Convert raw forward pass data into searchable graph.

File: `planning/graph_builder_tensor.py`

INPUT:

- `states_tensor`: (N, 8) unique states
- `graph_data`: Raw edges from forward pass

OUTPUT: CSR GRAPH DICTIONARY

```
graph = {
    'row_ptr': tensor([0, 5, 12, ...]),  # (N+1,) CSR row pointers
    'col_idx': tensor([3, 7, 15, ...]),  # (E,) target state indices
    'utilities': tensor([1.2, 0.8, ...]), # (E,) edge utilities
    'actions': tensor([[1,2,3], ...]),    # (E, 3) action data
    'state_time_offsets': {...}           # Time → index mapping
}
```

CSR Format Explained

CSR = Compressed Sparse Row

MEMORY LAYOUT:

```
States:   [S0]    [S1]    [S2]    [S3]    ...
           ↓       ↓       ↓       ↓
row_ptr:  [0,      3,      5,      5,      8, ...]
           |       |       |       |
col_idx:  [1,2,5, 3,4,    (none), 0,2,7, ...]
```

HOW TO FIND EDGES FROM STATE I:

```
start = row_ptr[i]
end = row_ptr[i + 1]
outgoing_edges = col_idx[start:end]  # O(1) lookup!
```

WHY CSR?

- **49.8% memory savings** vs COO (Edge List)
- **O(1) edge lookup** vs O(log E) binary search

CSR Memory Comparison

OUR GRAPH: 1.53M STATES, 326M EDGES

Format	Storage	Size
Dense Matrix	N × N floats	8.7 TB (impossible!)
COO (Edge List)	2 × E integers	2.6 GB
CSR	(N+1) + E integers	**1.3 GB**

CSR BREAKDOWN:

```
row_ptr:  (1,530,000 + 1) × 4 bytes = 6.1 MB
col_idx:  326,000,000 × 4 bytes     = 1,304 MB (1.27 GB)
utilities: 326,000,000 × 4 bytes    = 1,304 MB
-------------------------------------------
Total (col_idx only):                 1.3 GB
```

Graph Builder Implementation (1/2)

```python
def build_graph(self, states_tensor, graph_data):
    """
    Build CSR graph from forward pass results.
    Time: ~29s for 1.5M states, 326M edges
    """
    # 1. Create time → index mapping
    state_time_offsets = {}
    for t in unique_times:
        mask = (states_tensor[:, 0] == t)
        state_time_offsets[t] = (start, end)

    # 2. Concatenate all edges
    all_sources = torch.cat([...])  # Source state indices
    all_targets = torch.cat([...])  # Target state indices
    all_utilities = torch.cat([...])
```

Graph Builder Implementation (2/2)

```
# 3. Sort by source for CSR
sort_idx = torch.argsort(all_sources)
sorted_sources = all_sources[sort_idx]
sorted_targets = all_targets[sort_idx]

# 4. Build row_ptr array
row_ptr = torch.zeros(N + 1)
row_ptr = torch.bincount(sorted_sources, minlength=N)
row_ptr = torch.cumsum(row_ptr, dim=0)

return {'row_ptr': row_ptr, 'col_idx': sorted_targets, ...}
```

Phase 3: Backward Induction

Computing the Value Function

Phase 3: Backward Induction Overview

Goal: Compute V(s) for every state.

File: `planning/backward_induction_tensor.py`

THE BELLMAN EQUATION:

$$V(s) = \ln \sum_{a} \exp(u(s,a) + V(s'))$$

KEY INSIGHT:

- Terminal states: V = 0 (at home, mandatory complete)
- Work backwards from t=1440 to t=0
- Use CSR for O(1) edge lookup

Why Per-Home Backward Induction?

TERMINAL CONDITION DEPENDS ON HOME ZONE:

```
Agent from CZONE_1:
  Terminal = (t=1440, zone=CZONE_1, mandatory_done=True)
  V = 0 only if they returned to CZONE_1

Agent from CZONE_100:
  Terminal = (t=1440, zone=CZONE_100, mandatory_done=True)
  V = 0 only if they returned to CZONE_100
```

SAME GRAPH, DIFFERENT V VALUES:

- **Graph Structure**: Shared (determined by mandatory sequence)
- **Terminal States**: Different per home
- **V Values**: Must be computed separately

Backward Induction: Code (1/2)

```python
def run(self, states_tensor, graph, home_zone_id):
    """
    Solve Bellman equation for specific home zone.
    Time: ~2s for 1.5M states
    """
    N = len(states_tensor)
    V = torch.full((N,), float('-inf'), device=self.device)

    # 1. Identify terminal states for THIS home zone
    is_terminal = (
        (states_tensor[:, 0] == 1440) &      # End of day
        (states_tensor[:, 1] == home_zone_id) &  # At home
        (states_tensor[:, 7] >= mandatory_length) # Mandatory done
    )
    V[is_terminal] = 0.0
```

Backward Induction: Code (2/2)

```python
    # 2. Iterate backwards through time
    for t in reversed(range(0, 1440, 15)):
        # Get states at time t
        start, end = state_time_offsets[t]

        # For each state, compute logsumexp over outgoing edges
        V[start:end] = scatter_logsumexp(
            Q_values, source_indices
        )

    return V
```

LogSumExp: GPU Implementation (1/2)

THE MATH

$$V(s) = \log \sum_a \exp(Q(s, a))$$

NUMERICAL STABILITY (MAX TRICK)

```
def scatter_logsumexp(q_values, source_indices, N):
    # 1. Find max Q for each source (prevents overflow)
    q_max = scatter_max(q_values, source_indices)

    # 2. Stable exponential
    exp_q = torch.exp(q_values - q_max[source_indices])
```

LogSumExp: GPU Implementation (2/2)

```
# 3. Sum and log
sum_exp = scatter_add(exp_q, source_indices)
V = q_max + torch.log(sum_exp)

return V
```

PERFORMANCE

- Processes **326 million edges** in single GPU operation
- ~2 seconds per home zone

BI Memory Management

PER-HOME V TENSOR:

```
V tensor size: 1.53M states × 4 bytes = 6.1 MB
```

FOR 144 HOME ZONES:

```
Total V storage: 144 × 6.1 MB = 878 MB
```

STRATEGY:

```
V_tensors = {}  # Store on GPU during BI
for home in home_zones:
    V = solver.run(states, graph, home_zone_id=home.value - 1)
    V_tensors[home] = V  # Keep for simulation

# Peak GPU memory: Graph (1.3GB) + V (6MB) = ~1.4GB
```

BI Optimization Strategies

Tiered Approach for Scalability (Dec 2024)

The BI Bottleneck

PROBLEM: BI SCALES LINEARLY WITH HOME ZONES

Scenario	BI Runs	Time (est.)
2 homes	2	~3s
20 homes	20	~27s
144 homes	144	~200s
144 × 10 individual types	1,440	~33 min

SOLUTION: TIERED OPTIMIZATION

```
Tier 1: Basis Function     Tier 2: Multi-Fidelity     Tier 3: Neural
  (1-2 BI runs)              (15-37 BI runs)             (0 BI runs)
        ↓                           ↓                         ↓
  9.3× speedup                2.5× speedup                  Future
```

Tier 1: Basis Function Approximation 🏆

CLI flag: `--basis-function` (DEFAULT - Recommended)

KEY INSIGHT

V functions differ primarily by terminal location. Correct V from reference:

$$V(s, home) \approx V_{ref}(s) + \beta \times \Delta travel\_time$$

WHY LINEAR WORKS (R^2 = 0.98)

- Travel disutility ∝ β × travel_time
- V difference propagates linearly through Bellman recursion
- Only 1 reference BI run + fast analytical correction

Basis Function: Theoretical Derivation (1/2)

1. THE BELLMAN DIFFERENCE

If we have two home zones h_1, h_2, the value functions satisfy: $V(s, h_1) = \max_a\{U(s,a) + \gamma V(s', h_1)\}$
$V(s, h_2) = \max_a\{U(s,a) + \gamma V(s', h_2)\}$

Basis Function: Theoretical Derivation (2/2)

2. LINEAR PROPAGATION

The only difference between h_1 and h_2 is the terminal reward at $T = 1440$. If travel disutility is linear: $U_{travel} = \beta_{TT} \times TT(zone, home)$ Then the difference $\Delta V = V(s, h_1) - V(s, h_2)$ propagates linearly backwards:

$$\Delta V \approx \beta \times (TT(s.zone, h_2) - TT(s.zone, h_1))$$

3. VFA FORMULATION

This is a specific case of **Linear Value Function Approximation**:

$$\hat{V}(s, h) = V_{ref}(s) + \theta^\top \phi(s, h)$$

where $\phi(s, h) = TT(s.zone, h_{ref}) - TT(s.zone, h)$.

Basis Function: Implementation

```
# planning/traveltime_correction_bi.py
class TravelTimeCorrectionBI:
    def __init__(self, device='cuda'):
        self.beta = None  # Learned coefficient

    def train(self, states, graph, training_zones, od_lookup):
        """Train β via OLS on 8 diverse zones."""
        # Run full BI for 8 training zones
        V_true = {z: full_bi(z) for z in training_zones}

        # Fit: ΔV = β × Δtravel_time
        # Result: β ≈ 0.079 (value per minute)

    def predict(self, state, home_target):
        """O(1) prediction instead of O(|S|×|A|) BI."""
        delta_tt = od_lookup.get_tt(state.zone, ref) - \
                   od_lookup.get_tt(state.zone, home_target)
        return V_ref[state] + self.beta * delta_tt
```

Basis Function: Validation Results (1/2)

PERFORMANCE (144 HOME ZONES)

Zone	R^2	MAE
CZONE_32	0.990	1.05
CZONE_56	0.984	1.33
CZONE_54	0.995	0.66
CZONE_120	0.952	2.68
Average	**0.982**	**1.36**

Basis Function: Validation Results (2/2)

SPEEDUP

```
Full BI (144 zones): 186s
Basis Function:       20s  → 9.3× faster!
```

Usage: `python calibration/batch_simulate_universal.py --basis-function`

Tier 2: Multi-Fidelity BI

CLI flag: `--multifidelity`

CONCEPT

Full BI for representative subset, interpolate the rest.

Multi-Fidelity: Theoretical Basis

SURROGATE MODELING & MULTI-FIDELITY OPTIMIZATION

- **High-Fidelity Data**: Expensive "ground truth" (Full BI runs).
- **Low-Fidelity Data**: Inexpensive but less accurate approximations (Interpolation).
- **Goal**: Achieve high-fidelity accuracy using a small set of high-fidelity data by leveraging abundant low-fidelity information.

COMPOSITE NEURAL NETWORK (MENG & KARNIADAKIS 2019)

- **Structure**: Composed of three sub-networks:

  Low-Fidelity NN: Trained on low-fidelity data.

  Linear High-Fidelity NN: Discovers linear correlations.

  Non-linear High-Fidelity NN: Discovers non-linear correlations.
- **In DDCM**: We use inverse distance weighting (IDW) as the "low-fidelity" surrogate, corrected by the "high-fidelity" full BI samples.

WORKFLOW

External zones (123-144): Always full BI (different cities)

15 representatives: Stratified sampling across Bzones

Remaining zones: k-NN interpolation (OD travel time as distance)

```
# planning/multifidelity_bi.py
class MultiFidelityBI:
    def run(self, states, graph, home_zones, od_lookup):
        # Full BI for representatives + external
        V_high = {z: full_bi(z) for z in representatives}

        # Interpolate remaining zones
        corrector = MultiFidelityCorrector(device='cuda')
        corrector.fit(V_high, all_zones, od_lookup)
        for zone in remaining_zones:
            V[zone] = corrector.predict(zone, V_high)
```

BI Optimization Comparison

144 HOME ZONES BENCHMARK

Method	BI Runs	Time	Speedup	Accuracy
Full BI	144	186s	1×	100%
Basis Function 🏆	8+1	**20s**	**9.3×**	95.7%
Multi-Fidelity	37	74s	2.5×	99.6%

TOTAL PIPELINE (GRAPH BUILD + BI)

```
Graph Build:      60s (same for all methods)
BI (full):       186s → BI (basis function): 20s
─────────────────────────────────────────────
TOTAL:           246s → 80s = 3.1× faster overall
```

Basis Function vs Multi-Fidelity: Key Differences

Aspect	Basis Function	Multi-Fidelity
Speed	Very Fast (9.3×)	Moderate (2.5×)
Accuracy	High if linear assumption holds	Higher spatial accuracy
Training	Requires 8 training zones to learn β	No training; uses IDW interpolation
Minimum Homes	Works with any count	Needs ≥15 homes to be efficient
Best When	Zone attractiveness is stable	Complex non-linear spatial effects

Basis Function: Trade-offs (1/2)

WHY IT'S FAST

Uses a **linear correction**: $V(s, home) \approx V_{ref}(s) + \beta \cdot \Delta TT$

- Only **1 reference BI run** + fast analytical lookup
- O(1) per state, no iterative computation

TRAINING REQUIREMENT

- Must run **full BI for 8 diverse training zones** to learn β via OLS
- If zone attractiveness model changes significantly, **β must be re-trained**

Basis Function: Trade-offs (2/2)

LIMITATION: LINEAR ASSUMPTION

- Assumes value difference is linear in travel time
- May lose accuracy if zone attractiveness becomes highly non-linear
- **Future**: Add non-linearity via a small NN to learn the correction function

Multi-Fidelity: Trade-offs (1/2)

WHY IT'S MORE RELIABLE FOR SPATIAL ACCURACY

- Uses **Inverse Distance Weighting (IDW)** interpolation
- Can capture non-linear spatial patterns in value functions
- No linear assumption required

EFFICIENCY CONSIDERATION

- Requires running **full BI for N representatives** (external zones + samples)
- Benefit only appears when you have **many home zones** to interpolate
- **With 1 home zone**: No interpolation possible → Multi-Fidelity = Full BI (no speedup)
- **With 144 homes**: Run 37 full BI, interpolate 107 → 2.5× speedup

Multi-Fidelity: Trade-offs (2/2)

EXTERNAL ZONES (BZONE 13): ALWAYS FULL BI

- **Bzone 13** = Cities outside Higashihiroshima (CZONE_123 to CZONE_144)
- Different spatial scale → cannot interpolate with local zones
- Always run full BI for these 22 external zones

WHEN TO USE

- When zone attractiveness is complex or under active calibration
- When spatial non-linearities matter (e.g., CBD vs suburban patterns)

Phase 4: Simulation

Generating Agent Trajectories

Phase 4: Simulation Overview

Goal: Generate daily schedules for N agents.

File: `planning/simulate_batch_tensor.py`

FUNCTION: `SIMULATE_BATCH_TENSOR()`

```
plans = simulate_batch_tensor(
    initial_state_indices,  # (N,) starting state indices
    V_tensor,               # Value function for this home zone
    graph,                  # CSR graph
    all_states_tensor,      # State definitions
    manager,                # State encoder/decoder
    end_time=1440,
    random_seed=42
)
```

Output: List of DataFrames with agent trajectories

Simulation: Core Algorithm

```
def simulate_batch_tensor(initial_indices, V, graph, states):
    N = len(initial_indices)
    current_states = initial_indices.clone()
    trajectories = [current_states]

    while not all_terminal(current_states):
        # 1. Find outgoing edges using CSR (O(1) per state)
        edge_starts = graph['row_ptr'][current_states]
        edge_ends = graph['row_ptr'][current_states + 1]

        # 2. Compute choice probabilities
        Q = utilities + V[targets]
        P = softmax(Q)  # Per-action probabilities

        # 3. Sample next action for ALL agents at once
        next_actions = torch.multinomial(P, num_samples=1)

        # 4. Update current states
        current_states = targets[next_actions]
        trajectories.append(current_states)

    return decode_trajectories(trajectories, states)
```

Parallel Sampling

TRADITIONAL (SEQUENTIAL):

```
for agent in range(N):
    action = sample_from(probabilities[agent])  # N loops!
```

GPU PARALLEL:

```
# Sample for ALL 100,000 agents in ONE operation
actions = torch.multinomial(P, num_samples=1)  # O(1) time!
```

PERFORMANCE:

Agents	Time
1	~0.2s
10	~0.2s
100	~0.4s

Universal Graph Approach

24.7× Performance Improvement (Dec 2024)

Universal Graph: Theoretical Basis (1/2)

STATE AGGREGATION & BISIMULATION

- **Core Idea**: Merge states that have identical future dynamics.
- **Probabilistic Bisimulation**: Two states s_1, s_2 are bisimilar if they have the same transition probabilities and rewards for all actions.
- **In DDCM**: Once an agent leaves home, their future options (activities, travel) depend on their current state, NOT their starting home zone.

Universal Graph: Theoretical Basis (2/2)

WHY IT WORKS

- The "History" state component tracks mandatory activity progress.
- Once the mandatory sequence is fixed, the transition graph is identical for all agents.
- **Result**: We can compute the graph ONCE and share it across all 144 home zones.

> !IMPORTANT**Current Scope**: "Universal" means shared across **home zones**, not work/school zones. Agents must share the same `mandatory_sequence` (Work Zone, School Zone) to share a graph. **Future Exploration**: Extending graph universality to work/school zones (one graph for all destinations).

Universal Graph: State Convergence

EMPIRICAL OBSERVATION

States from different homes converge rapidly as the day progresses.

Time	Unique States (1 Home)	Unique States (144 Homes)
0:00	1	144
5:00	~4,000	~4,500
10:00	~12,000	~12,200
14:00	~24,000	~24,400

THEORETICAL JUSTIFICATION

The Problem: Per-Home Graphs

OLD ARCHITECTURE:

```
For 144 home zones × 1 mandatory sequence:

  Group 1 (Home=1):   Forward 30s + Graph 29s + BI 2s = 61s
  Group 2 (Home=2):   Forward 30s + Graph 29s + BI 2s = 61s
  ...
  Group 144 (Home=144): Forward 30s + Graph 29s + BI 2s = 61s

  TOTAL: 144 × 61s = 8,784s (2.4 hours!)
```

THE WASTE:

- Each graph has ~1.5M states
- But states OVERLAP heavily between homes
- We’re re-building the same graph 144 times!

The Solution: Universal Graph

NEW ARCHITECTURE:

```
For 144 home zones × 1 mandatory sequence:

  ONCE: Universal Forward Pass (144 origins) = 30s
  ONCE: Graph Build = 29s

  BI for Home 1: 2s
  BI for Home 2: 2s
  ...
  BI for Home 144: 2s

  TOTAL: 30s + 29s + (144 × 2s) = 357s (6 minutes!)
```

SPEEDUP: **24.7× FASTER!**

Why It Works: State Deduplication

AFTER LEAVING HOME:

```
t=0 (Start):     144 unique states (one per home)
t=300 (5:00 AM): 4,503 states (paths start merging)
t=600 (10:00):   12,236 states (heavy overlap)
t=840 (2:00 PM): 24,454 states (almost identical)
t=1200 (8:00 PM): 27,360 states (fully merged)
```

STATE DEFINITION:

```
State = (time, current_zone, activity, duration, mode, ...)
```

Your starting home doesn't affect states once you've left!

Universal Graph: Scaling Results

BENCHMARK: 1 MANDATORY SEQUENCE × N HOME ZONES

Homes	States	Graph Build	BI Total	**TOTAL**	Speedup
1	1.52M	60s	1.3s	61s	1×
10	1.53M	58s	13s	71s	**8.3×**
50	1.53M	58s	103s	162s	**18.7×**
100	1.53M	59s	225s	284s	**21.5×**
144	1.54M	59s	298s	**357s**	**24.7×**

Key Insight: States stay ~1.53M regardless of home count!

Memory Footprint

UNIVERSAL GRAPH (144 HOMES):

Component	Size	Notes
States Tensor	48 MB	1.54M × 8 × 4 bytes
CSR Graph	1.3 GB	row_ptr + col_idx
V Tensors (×144)	878 MB	144 × 6.1 MB
Total	**~2.2 GB**	Fits GTX 1080 Ti (11GB)

COMPARISON:

```
Old approach: 144 × 1.3 GB = 187 GB (IMPOSSIBLE!)
New approach: 1.3 GB + 0.9 GB = 2.2 GB ✓
```

Implementation: Key Functions

`RUN_UNIVERSAL_FORWARD_PASS()`

```
states, graph_data, home_indices, stats = run_universal_forward_pass(
    od_lookup, zone_attr,
    initial_states,      # List[State] - 144 homes!
    has_child, mandatory_sequence,
    device='cuda'
)
# home_indices = {Zone.CZONE_1: 0, Zone.CZONE_2: 1, ...}
```

`BACKWARDINDUCTION.RUN()` (PER HOME)

```
for home in home_zones:
    V = solver.run(
        states, graph,
        home_zone_id=home.value - 1  # Different terminal!
    )
```

Bug Fixes (December 2024)

BUG 1: MISSING GPU TRANSFER

```
# BEFORE (100× slower!)
current_batch = torch.cat(tensors_at_t, dim=0)

# AFTER
current_batch = torch.cat(tensors_at_t, dim=0)
if current_batch.device.type != self.device:
    current_batch = current_batch.to(self.device)  # Critical!
```

BUG 2: MISSING STATE PRUNING

```
# BEFORE (no pruning)
pass

# AFTER (added to run_multi_origin)
current_batch = self._prune_unreachable_states(current_batch, t)
```

Both bugs were in `run_multi_origin()`, not `run()`!

Academic Context & Research Gaps

Grounding the Work in Existing Theory

Research Gap Analysis

1. THE "INTEGRATION" GAP

Literature treats "Discrete Choice" and "Reachability Analysis" as separate fields.

- **Our Contribution**: Deeply integrated methodology using Reachability (Forward Reachable Tube) to rigorously define the choice set for DCM.

2. THE "PRUNING" GAP

State Space Pruning in ABMs is often an ad-hoc heuristic.

- **Our Contribution**: Formalizing pruning using **Conditions** (Logic/Constraints), moving from "hacking" to mathematical formulation.

Research Gap Analysis (cont.)

3. THE "BISIMULATION IN TRANSPORT" GAP

Bisimulation Metrics are common in AI/RL but sparse in Transportation.

- **Our Contribution**: Universal Graph as **State Aggregation via Bisimulation**, introducing rigorous AI concepts to the Transportation domain.

4. THE "BASIS FUNCTION" GAP

Linear VFA is common in RL, but novel for **spatial transferability** of Value Functions between different home locations in a city graph.

Key Theoretical References

- **Viability Theory**: Aubin (1991), Coquelin & Munos (2007)
- **Reachability Analysis**: Mitchell et al. (2005), Althoff (2021)
- **Time Geography**: Hägerstrand (1970), Miller (2005)
- **Approximate DP**: Powell (2007), Bertsekas (2012)
- **Multi-Fidelity**: Meng & Karniadakis (2019)

Performance Summary

Benchmarks & Comparisons

Full Pipeline Timing

SINGLE MANDATORY SEQUENCE × 144 HOME ZONES

Phase	Full BI	With Basis Function 🏆
Data Loading	~3s	~3s
Reachability Masks	~0.2s	~0.2s
Forward Pass	**30s**	**30s**
Graph Build	**29s**	**29s**
BI (×144)	**186s**	**20s** (9.3× faster)
Simulation	~5s	~5s
TOTAL	**~253s**	**~87s**

Hardware Requirements

MINIMUM:

- GPU with 4GB VRAM (single home zone)
- 16GB system RAM

RECOMMENDED (FULL SCALE):

- **GPU**: 8GB+ VRAM (GTX 1080, RTX 2070, etc.)
- **Tested**: GTX 1080 Ti (11GB VRAM)

PEAK MEMORY USAGE:

```
Forward Pass: ~3GB GPU
Graph Build:  ~2GB GPU (sorting overhead)
BI + Sim:     ~2GB GPU
```

Key Files Reference

File	Purpose
`calibration/batch_simulate_universal.py`	Universal Graph batch sim
`planning/forward_pass_tensor.py`	`run_universal_forward_pass()`
`planning/graph_builder_tensor.py`	CSR graph construction
`planning/backward_induction_tensor.py`	Value function solver
`planning/traveltime_correction_bi.py`	**Tier 1: Basis Function (9.3×)**
`planning/multifidelity_bi.py`	**Tier 2: Multi-Fidelity (2.5×)**
`planning/simulate_batch_tensor.py`	Parallel agent simulation

Thank You

DDCM GPU Implementation

High-Performance Activity-Based Modeling

Universal Graph: 24.7× Speedup | **Basis Function BI**: 9.3× Speedup

Combined: ~100× faster than original approach

Powered by PyTorch & CUDA | CSR Sparse Graphs | Level-Synchronous BFS