DDCM GPU v1 Literature Review

High-Performance Discrete Choice Models

Press Space for next page →

The Problem

Computational Cost in Recursive Logit

The Recursive Logit Problem

Recursive Logit (RL) models suffer from severe computational limitations when scaled.

THE CORE BOTTLENECK

The typical implementation requires solving a large system of linear equations:

```
Z <- solve(I - M) %*% b
```

- `M`: Transition matrix (State $\rightarrow$ State).
- `solve()`: Requires inverting `(I-M)`.

The Dense Matrix Trap

Even if `M` is sparse, the inverse `(I-M)^-1` is **Dense**.

WHY THIS KILLS PERFORMANCE

- **Memory**: Storing a dense $N \times N$ matrix for large N is impossible.
- **Computation**: Inversion is $O(N^3)$.

This approach becomes computationally prohibitive as the state space grows.

DDCM & Curse of Dimensionality

Dynamic Discrete Choice Models (DDCM) extend RL to daily activity patterns, exploding the state space.

9 Dimensions of Complexity

A state isn't just a location. It's a tuple:

- **Location** ($N_L \approx 1240$)
- **Mode** ($N_M \approx 4$)
- **Purpose** ($N_P \approx 6$)
- **Time** ($T \approx 96$)
- **History** ($H \approx 2$)
- ...and more.

The Naive State Space

If we pre-allocated every possible state combination:

$$S_{naive} = N_L \times N_M \times N_P \times T \times \dots$$

$$1240 \times 4 \times 6 \times 96 \times \dots \approx \mathbf{Billions}$$

THE IMPLICATION

We are trying to solve a system with **Billions of Unknowns**.

The Real-World Cost: Memory

Why we can’t just "buy a bigger computer".

1. MEMORY EXPLOSION

Storing the utility array alone:

```
u <- array(0, dim=c(N_L, N_M, N_P, T, ...))
```

* For a realistic city (Stockholm): **Terabytes of RAM**.

The Real-World Cost: Time

2. TIME TO ESTIMATE

- Västberg et al. (2020) reported **4-10 seconds per individual** for a toy network.
- For a full city? **1,000+ days** on a single core.

3. THE SPARSE-TO-DENSE TRAP

We start with sparse data (Road Network), but the math (`solve`) forces us into dense matrices, negating all efficiency.

Our Objective: Current State

THE PROBLEM

- Intractable for large cities
- Requires massive clusters
- Slow iteration cycles

Our Objective: The Goal

TARGET OUTCOME

- **Reduce Computational Burden** by orders of magnitude
- **Explore New Logic**: Shift from Linear Algebra to Graph Theory
- **Enable Real-Time** estimation on consumer hardware

The Insight

Reachability & Condition-Based Modeling

Inspiration: Hamilton-Jacobi Reachability

We draw inspiration from **Control Theory** and **Robotics** (Doshi et al., 2022).

THE CONCEPT

In robotics, we don’t calculate the path for *every* point in the universe. We calculate a **Reachability Tube**:

- **Backward Reachable Tube (BRT)**: All states that *can* reach the target.
- **Forward Reachable Tube (FRT)**: All states reachable from the start.

> "Don't solve for the world. Solve for the Tube."

Condition-Based Pruning

Applying Reachability to Travel Demand.

THE REALITY

Individuals do not consider the full state space.

- **Time**: You can't be at work at 9:00 and home at 9:05 if it takes 30 mins.
- **Geography**: You won't walk 50km for a lunch break.
- **Logic**: You can't drive if you didn't bring your car.

THE RESULT

By applying these **Conditions** during state generation, we prune the state space:

$$S_{reachable} \ll S_{naive}$$

Reduction Factor: ~99.99%

Back to Basics

Algorithms & Complexity

What is an Algorithm?

> "Any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output." — *Cormen et al., Introduction to Algorithms*

A sequence of computational steps that transform the input into the output.

Big O Notation

Why do we care about complexity?

Big O Notation describes the **Upper Bound** of an algorithm's running time in the **Worst-Case Scenario**. It tells us how the runtime grows as input size (N) increases.

WHY IT MATTERS FOR DDCM

- **Input (N)**: Number of Agents (1,000 - 100,000+)
- **State Space (S)**: Potentially Billions.

If our algorithm is $O(N^2)$ or $O(S^2)$, we are dead. We need $O(N)$ or $O(S_{reachable})$.

Computational Complexity

CPU (Sequential) vs **GPU (Parallel)**

SEQUENTIAL CPU

Process states one by one.

O(N × B)

- N: Total States (10^6)
- B: Branching Factor (50)
- **Time**: ~60 seconds

PARALLEL GPU

Process entire time layers.

O(T)

- T: Time Steps (96)
- Independent of N (until saturation)
- **Time**: ~3 seconds

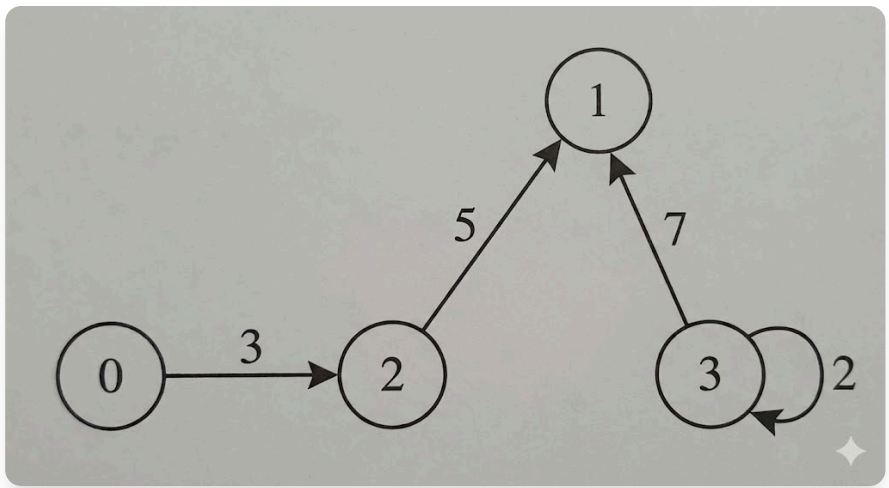
Graph Theory

Traversing the State Space

Graph Algorithm

A **Graph** $G = (V, E)$ models relationships.

- **Vertices (V)**: Nodes (e.g., States in our DDCM: `Time=9:00, Zone=Home`).
- **Edges (E)**: Connections (e.g., Transitions: `Travel to Work`).

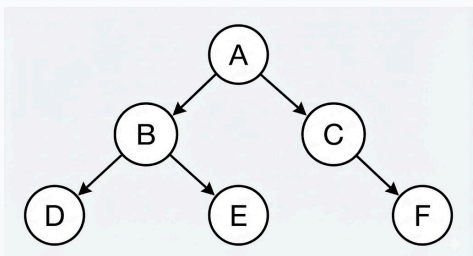


Graph Traversal Overview

How we explore the state space.

BFS (BREADTH-FIRST SEARCH)

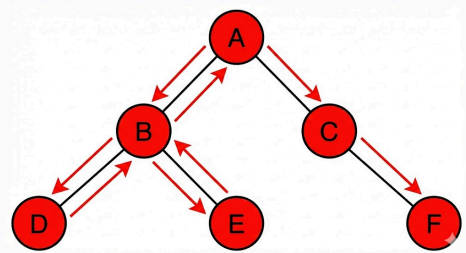
Explores layer-by-layer.



"The Ripple"

DFS (DEPTH-FIRST SEARCH)

Explores deep into one path.



"The Maze"

BFS: Definition & Goal

Breadth-First Search (BFS) is a systematic way to explore a graph.

THE "WAVE" PROPERTY

It discovers vertices in **waves** emanating from the source s:

Distance 0: The source s.

Distance 1: Direct neighbors.

Distance 2: Neighbors of neighbors.

THE GOAL

To find the **Shortest Path Distance** $\delta(s, v)$ from s to every reachable vertex v.

- In an **Unweighted Graph**, this is the path with the **fewest edges**.

BFS: Unweighted Graph Context

Our Forward Pass operates on an **Unweighted Graph** (Time).

WHAT IS AN UNWEIGHTED GRAPH?

- Edges have no numerical value (or implicitly weight = 1).
- **Shortest Path** = Minimum number of steps.

CONTRAST

Feature	Unweighted Graph	Weighted Graph
Edge Value	Unit (1)	Real number (Cost, Time)
Shortest Path	Fewest Edges	Lowest Total Weight
Algorithm	**BFS**	Dijkstra

BFS: Core Mechanism

How BFS maintains order.

1. THE FIFO QUEUE

- **First-In, First-Out**.
- Ensures nodes are processed in the exact order they are discovered.
- Maintains the "Wave" structure (Layer k before Layer $k+1$).

2. VERTEX COLORING (STATE TRACKING)

To prevent cycles and redundant work:

- **WHITE**: Undiscovered.
- **GRAY**: Discovered, in Queue (Frontier).
- **BLACK**: Finished (Neighbors explored).

BFS: Pseudocode

```
def BFS(graph, source):
    # Initialize
    queue = [source]
    visited = {source}

    while queue:
        u = queue.pop(0)  # Dequeue (FIFO)

        for v in graph.neighbors(u):
            if v not in visited:  # Check Color (White?)
                visited.add(v)    # Mark Gray
                v.distance = u.distance + 1
                queue.append(v)   # Enqueue

        # u is now Black
```

DFS: Depth-First Search

The "Maze Solver" approach.

- **Structure**: **Stack** (LIFO - Last-In, First-Out).
- **Behavior**: Goes as deep as possible down one path before backtracking.
- **Use Case**: Cycle detection, topological sorting.

WHY NOT FOR US?

- **Sequential**: Hard to parallelize (one path at a time).
- **Not Level-Synchronous**: Doesn’t naturally fit our "Time Step" physics.

Advanced Data Structures

CSR & Memory Efficiency

What is CSR?

Compressed Sparse Row is a memory-efficient way to represent an **Adjacency Matrix**.

1. THE ADJACENCY MATRIX

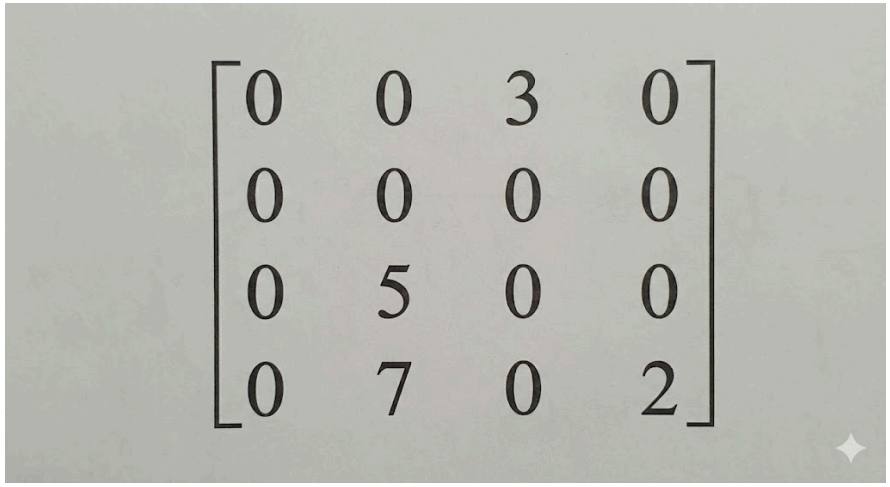
A simple grid where:

- **Rows** = Source Nodes
- **Columns** = Destination Nodes
- **Value** = Edge Weight (or 0 if no edge)

THE PROBLEM

- **Space**: $O(N^2)$. For 50,000 states, we need **2.5 Billion** cells.
- **Waste**: Our graph is **Sparse**. 99% of these cells are **Zero**.

Adjacency Matrix Visual



4×4 Adjacency Matrix

2. THE CORRELATION

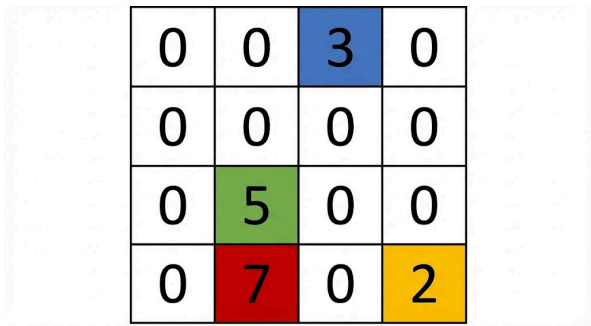
CSR is just the Adjacency Matrix without the Zeros. Instead of storing the full grid, we flatten the non-zero values into 3 compact arrays.

Compressed Sparse Row (CSR)

Stores only non-zero values using 3 arrays.

1. THE ORIGINAL SPARSE MATRIX

Most elements are zero. We only care about the colored numbers.



THE 3 ARRAYS

Values: `[3, 5, 7, 2]` (The non-zero numbers)

Column Indices: `[2, 1, 1, 3]` (Column for each value)

CSR Step-by-Step: The Logic

How do we build the `Row Pointers`?

Think of it as a set of **bookmarks** that tell us where each row starts in the `Values` array.

THE CALCULATION (CUMULATIVE SUM)

Start: Always `[0]`.

After Row 0 (1 item): `0 + 1 = 1` → `[0, 1]`

After Row 1 (0 items): `1 + 0 = 1` → `[0, 1, 1]` *(Repeated = Empty Row!)*

After Row 2 (1 item): `1 + 1 = 2` → `[0, 1, 1, 2]`

After Row 3 (2 items): `2 + 2 = 4` → `[0, 1, 1, 2, 4]`

The final number (4) is the total count of non-zero items.

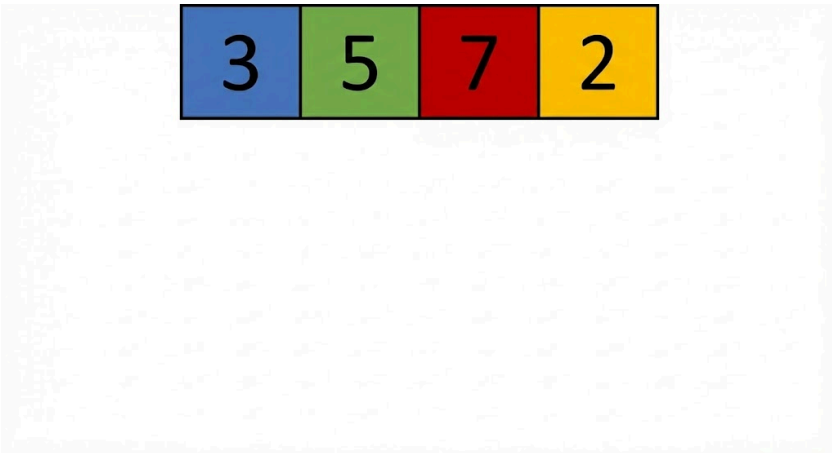
Visualizing CSR Ranges

The `Row Pointers` define the **Start** and **End** index for every row.

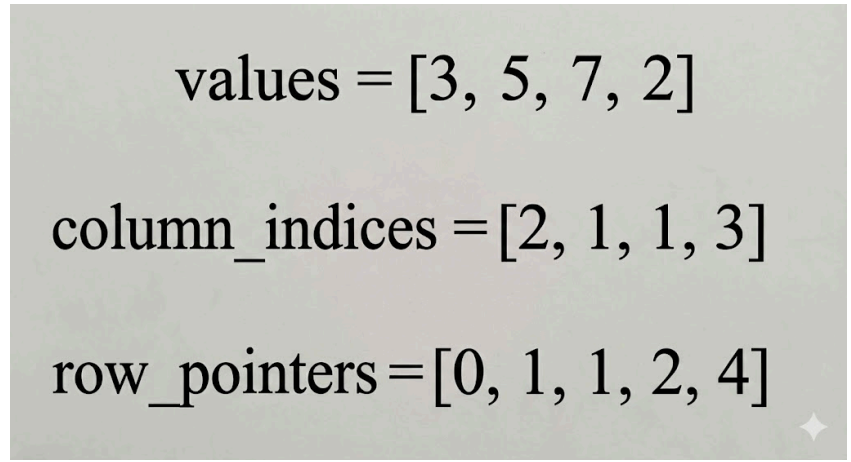
Row	Start Index	End Index	Calculation	Items
Row 0	**0**	**1**	`1 - 0`	1 item
Row 1	**1**	**1**	`1 - 1`	0 items (Empty)
Row 2	**1**	**2**	`2 - 1`	1 item
Row 3	**2**	**4**	`4 - 2`	2 items

CSR Visuals

Mapping the values to the arrays.



Values Array



CSR Representation

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 state space is a **Directed Acyclic Graph (DAG)** in time. As paths merge at common locations/activities, the "memory" of the starting home zone is lost in the state representation, leading to

Literature Comparison

Why Level-Synchronous BFS?

Paper 1: Tithi et al. (2022)

Optimal Level-Synchronous Parallel BFS

OVERVIEW

This paper introduces **S3BFS**, a work-adaptive parallel BFS algorithm. It dynamically adjusts the number of active cores based on the workload at each level to save energy without sacrificing performance.

Paper 1: Connection to DDCM

SIMILARITIES

- **Level-Synchronous**: Like S3BFS, our Forward Pass processes states level-by-level (Time Steps).
- **Deduplication**: Both require handling duplicate states (multiple paths to the same node).

DIFFERENCES

- **Core Count**: S3BFS changes core count dynamically. We use **Fixed Occupancy** (max GPU cores) because our goal is pure speed, not energy efficiency.
- **Prefix Sum**: They use explicit prefix sums for work distribution. We use PyTorch's implicit schedulers.

Paper 2: Berrendorf et al. (2014)

Level-Synchronous BFS for Multicore/Multiprocessor

OVERVIEW

Evaluated 6 BFS algorithms on NUMA (Non-Uniform Memory Access) systems. Found that **Container-Centric** approaches (using explicit queues) outperform **Vertex-Centric** ones (checking all nodes).

Paper 2: Connection to DDCM

KEY TAKEAWAYS

- **Container-Centric**: We use explicit "Frontier Tensors" (queues) to store active states, just like their best-performing algorithms (`socketlist` , `bitmap`). We do *not* iterate over all possible states (Vertex-Centric).
- **Avoiding Atomics**: They showed atomic operations kill performance >16 cores. We avoid fine-grained atomics by using **Global Synchronization** (PyTorch barriers) and **Batch Operations**.
- **Deduplication**: They use a bitmap for visited checks. We use `torch.unique` (hash-based sort) which serves the same purpose but is optimized for GPU.

Paper 3: Merrill et al. (2012)

Scalable GPU Graph Traversal

OVERVIEW

The seminal paper on GPU BFS. Introduced **Parallel Prefix Sum** for conflict-free queue insertion and **Three-Tier Expansion** (Scan, Warp, CTA) to handle varying node degrees.

Paper 3: Connection to DDCM

STRONGEST MATCH

- **Batched Processing**: Merrill processes vertices in batches (CTAs). We process agents in batches (Tensor Batches).
- **Two-Phase Process**:

  Expansion: Generate neighbors (Edge Frontier).

  Contraction: Filter and Deduplicate (Vertex Frontier).

  - *We follow this exact pattern in* `forward_pass_tensor.py`.

- **Deduplication**: Merrill proved local culling removes 95% of duplicates. We currently rely on global `torch.unique`, which is our main area for future optimization (adopting their Warp Culling).

Backward Induction Details

The Mathematical Engine.

The Bellman Equation

We solve a **Finite Horizon MDP** with **Gumbel-distributed noise**.

$$V(s) = \ln \left(\sum_{a \in Actions} \exp(\underbrace{u(s,a) + V(s')}_{Q(s,a)}) \right)$$

- $u(s,a)$: Immediate Utility.
- $V(s')$: Value of future state.
- **LogSumExp**: Acts as a "Soft Max".

THE NUMERICAL PROBLEM

Directly computing $\exp(Q)$ is dangerous.

- $\exp(1000) \to \infty$ (Overflow).

Scatter LogSumExp: The Problem

THE GOAL

Compute the "Soft Max" value for each state:

$$V(s) = \ln \sum \exp(Q(s, a))$$

THE NUMERICAL DANGER

Directly computing $\exp(Q)$ is dangerous because exponentials grow too fast.

- $Q = 1000 \implies \exp(1000) = \infty$ (Overflow)
- $Q = -1000 \implies \exp(-1000) = 0$ (Underflow)

Scatter LogSumExp: The Solution

"THE MAX TRICK"

We shift the values by subtracting the maximum Q for each group.

$$\log \sum \exp(x_i) = x_{max} + \log \sum \exp(x_i - x_{max})$$

Scatter LogSumExp: Visual Example (1/3)

Let's trace the data for **2 States** and **3 Edges**.

INPUT DATA

- **Edges (Q-Values)**: `[10.0, 12.0, 20.0]`
- **Source Indices**: `[0, 0, 1]`
  - *Edges 0 & 1 belong to State 0.*
  - *Edge 2 belongs to State 1.*

STEP 1: SCATTER MAX

Find the max Q for each Source ID.

- **State 0**: $\max(10, 12) = 12$
- **State 1**: $\max(20) = 20$
- **Max Tensor**: `[12.0, 20.0]`

Scatter LogSumExp: Visual Example (2/3)

STEP 2: STABLE EXPONENTIALS

Subtract the Max from each Edge and exponentiate.

- Edge 0: $\exp(10 - 12) = e^{-2} \approx 0.135$
- Edge 1: $\exp(12 - 12) = e^{0} = 1.0$
- Edge 2: $\exp(20 - 20) = e^{0} = 1.0$
- **Exp Tensor**: `[0.135, 1.0, 1.0]`

Scatter LogSumExp: Visual Example (3/3)

STEP 3: SCATTER SUM

Sum the stable values for each Source ID.

- **State 0**: $0.135 + 1.0 = 1.135$
- **State 1**: 1.0

STEP 4: FINAL LOG

Add the Max back to the log of the sum.

- **State 0**: $12 + \ln(1.135) \approx 12 + 0.127 = \mathbf{12.127}$
- **State 1**: $20 + \ln(1.0) = 20 + 0 = \mathbf{20.0}$

Optimization Theory

Basis Functions & Multi-Fidelity

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:

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

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

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.

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)

Thank You

High-Performance DDCM Implementation

Powered by PyTorch & CUDA