

DDCM GPU v1 Implementation

High-Performance Discrete Choice Models

Press Space for next page →

Basic Definition

Foundational concepts for our implementation.

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 Problem

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)**

THE SCALE PROBLEM (NAIVE APPROACH)

If we pre-allocated every possible state combination:

$$S_{naive} = |T| \times |Z| \times |A| \times |D| \times |M| \times \dots$$

$$96 \times 100 \times 10 \times 20 \times 5 \approx \mathbf{1.92 \times 10^9} \text{ states}$$

The Solution: Reachability

We don't store everything. We only store what is **Reachable**.

$$S_{reachable} = \{s \in S_{naive} \mid \exists \text{ valid path } s_0 \rightarrow \dots \rightarrow s\}$$

THE REDUCTION FACTOR

By applying constraints (Time, Geography, Logic) *during* generation:

$$S_{reachable} \approx \mathbf{10^5} \text{ states}$$

Reduction: $\frac{10^9}{10^5} = 10,000 \times (\text{99.99\% Pruned})$

Computational Complexity

CPU (Sequential) vs GPU (Parallel)

SEQUENTIAL CPU

Process states one by one.

$\$O(N \times B)$

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

PARALLEL GPU

Process entire time layers.

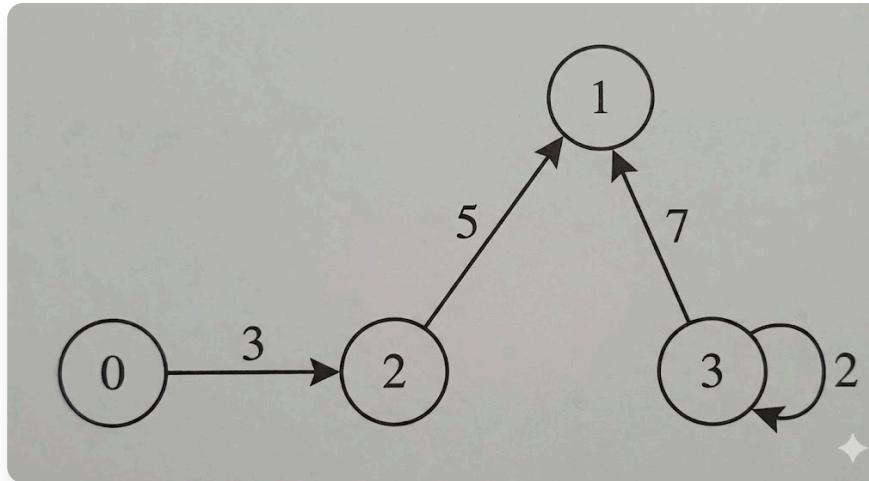
$\$O(T)$

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

Graph 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).

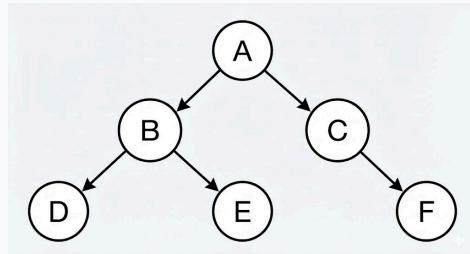


Graph Traversal Overview

How we explore the state space.

BFS (BREADTH-FIRST SEARCH)

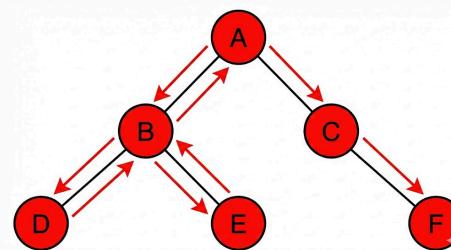
Explores layer-by-layer.



"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.

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

$$\begin{bmatrix} 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 7 & 0 & 2 \end{bmatrix}$$

4x4 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.

0	0	3	0
0	0	0	0
0	5	0	0
0	7	0	2

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): $0 + 1 = 1 \rightarrow [0, 1]$

After Row 1 (0 items): $1 + 0 = 1 \rightarrow [0, 1, 1]$ (*Repeated = Empty Row!*)

After Row 2 (1 item): $1 + 1 = 2 \rightarrow [0, 1, 1, 2]$

After Row 3 (2 items): $2 + 2 = 4 \rightarrow [0, 1, 1, 2, 4]$

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

values = [3, 5, 7, 2]

column_indices = [2, 1, 1, 3]

row_pointers = [0, 1, 1, 2, 4]

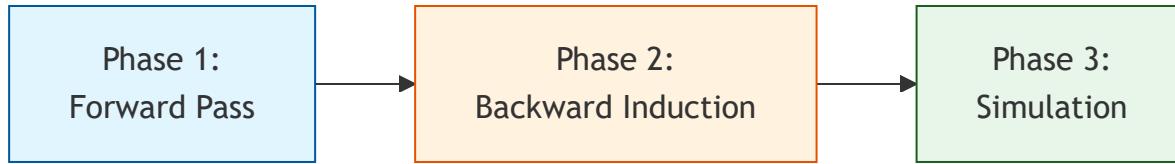
CSR Representation

Our Implementation

Applying these concepts to High-Performance GPU Computing.

Full Algorithm Overview

Three distinct phases orchestrated by `main.py`.



We will explore each phase in the following slides.

Phase 1: Forward Pass (Reachability)

Goal: Discover all reachable states (S) and construct the State Space.

- **Input:** Initial State + Constraints (Timetables, Opening Hours).
- **Output:** A set of unique states reachable at every time step t .
- **Graph Type:** Unweighted (Time-based).
- **Algorithm:** Level-Synchronous BFS with Time Buckets.

WHY TENSOR-BASED?

- **Original:** Python `dict` + `set`. Slow object overhead.
- **Optimized:** PyTorch Tensors. Processes thousands of states in parallel.

Phase 1: Step A - Gather & Merge

File: `planning/forward_pass_tensor.py`

At each time step t , we collect and deduplicate states.

```
# 1. GATHER: Pop all state batches scheduled for time t
tensors_at_t = pending_states.pop(t, [])
current_batch_raw = torch.cat(tensors_at_t, dim=0)

# 2. MERGE: Deduplicate to prevent explosion
# Uses torch.unique to find unique paths
current_batch_unique = self.manager.deduplicate_tensors(current_batch_raw)

# Store for graph building
all_states_tensors.append(current_batch_unique)
```

Key Insight: Deduplication reduces the state space by merging identical paths (e.g., arriving at the same location at the same time via different routes).

Phase 1: Step B - Expand & Scatter

File: `planning/forward_pass_tensor.py`

We generate next states using a GPU Kernel and schedule them.

```
# 3. EXPAND: Generate all valid next states in parallel
next_states, utilities, ... = self.kernel.expand_states_batch_tensor(
    current_batch_unique, ...
)

# 4. SCATTER: Group by arrival time
next_times = next_states[:, 0]
unique_times = torch.unique(next_times)

for next_t in unique_times:
    # Schedule for future processing
    mask = (next_times == next_t)
    pending_states[next_t.item()].append(next_states[mask])
```

Phase 2: Backward Induction

Goal: Compute the Value Function $V(s, t)$ for every state.

- **Input:** The State Space from Phase 1.
- **Output:** A dense tensor V where $V[i]$ is the max expected utility from state i .
- **Graph Type:** Weighted (Utility-based).
- **Algorithm:** Bellman Equation ($V(s) = \ln \sum \exp(u + V_{next})$).

THE CHALLENGE

The Forward Pass gives us a "Raw Pile" of transitions. We need a structured graph to propagate values backwards.

Phase 2: Step A - Graph Building

File: `planning/graph_builder_tensor.py`

We transform the raw transitions into a **Global Indexed Graph**.

Global Indexing: Assign a unique ID ($0 \dots N$) to every state.

Edge Mapping: Convert "State Content" edges to "Index" edges ($ID_A \rightarrow ID_B$).

Sorting: Sort all edges by `SourceID`.

Why Sort? It allows us to use **Binary Search** to find edges during the backward pass, avoiding the need for a massive adjacency matrix.

Phase 2: Step B - Vectorized Pass

File: `planning/backward_induction_tensor.py`

We solve the Bellman Equation for millions of edges at once.

```
# 1. Slice Edges for time t (Binary Search)
start = torch.searchsorted(edge_sources, state_start)
end   = torch.searchsorted(edge_sources, state_end)

# 2. Compute Q-Values (Vectorized)
# Q(s,a) = Utility + V(next)
q_values = utilities[start:end] + V[targets[start:end]]

# 3. Aggregation (Scatter LogSumExp)
# V(s) = log(sum(exp(Q)))
values = scatter_logsumexp(q_values, sources[start:end] - state_start)
```

Phase 3: Simulation

Goal: Generate daily schedules for N agents.

- **Input:** Initial States + Value Function V .
- **Output:** Trajectories (Sequence of states).
- **Algorithm:** Probabilistic Batch Sampling.

BATCH VS. SINGLE

- **Single-Agent:** Loop N times. Slow Python overhead.
- **Batch Simulation:** Simulate 100,000 agents **simultaneously** as a single tensor operation.

Phase 3: The Simulation Loop

File: `planning/simulate_batch_tensor.py`

```
# Initialize N agents
current_indices = initial_indices.clone()

for step in range(max_steps):
    # 1. Lookup Edges for ALL agents
    starts = torch.searchsorted(edge_sources, current_indices)
    ends   = torch.searchsorted(edge_sources, current_indices + 1)

    # 2. Compute Probabilities
    # P(a) ~ exp(Q(s,a))
    q_values = utilities[edge_indices] + V[targets]
    probs = torch.softmax(q_values, dim=1)
```

Phase 3: Sampling & Update

File: `planning/simulate_batch_tensor.py`

We sample actions and move agents in parallel.

```
# 3. Sample Actions (Monte Carlo)
# Returns one action index per agent
samples = torch.multinomial(probs, 1)

# 4. Update States
# Gather the Target Node IDs based on selected actions
selected_targets = targets.gather(1, samples)

# Move agents
current_indices = selected_targets
```

Result: 100,000 agents move one step forward in milliseconds.

Deep Dive: CSR Implementation

File: 0 - Inbox/drafts/full algorithm/csr_graph_structure.md

How we store the graph efficiently.

THE DATA TRANSFORMATION PIPELINE

Stage A: The "Raw Pile" (COO)

- *During Forward Pass.* Edges are appended in random discovery order. Fast write, impossible to search.

Stage B: The "Assembly" (Global Indexing)

- *Graph Builder.* Assign IDs, resolve targets. Still unsorted.

Stage C: The "Implicit CSR" (Sorted)

- *Graph Builder.* Sort by SourceID. This enables binary search lookup.

Implicit vs. Explicit CSR

Why we chose "Implicit" (Sorted Edge List) over Standard CSR (`row_ptr`).

Feature	Standard CSR	Our "Implicit CSR"
Structure	<code>row_ptr</code> , <code>cols</code> , <code>vals</code>	<code>sources</code> (sorted), <code>targets</code> , <code>vals</code>
Build Operation	Histogram + Prefix Sum	Global Sort (<code>torch.sort</code>)
Build Cost	Complex on GPU	Very Fast (Optimized Primitive)
Lookup	Array Access ($O(1)$)	Binary Search ($O(\log E)$)

Trade-off: We sacrifice a tiny bit of lookup speed ($O(\log E)$ vs $O(1)$) for **massive gains in build speed**, which is critical since we rebuild the graph frequently.

CSR Visual Example

Step 1: Raw Collection (COO) `[(0->2, 10), (1->2, 20), (0->1, 5)]` (Random Order)

Step 2: Sorting (Our Format) Sort by Source:

- Index 0: `0 -> 2 (10)`
- Index 1: `0 -> 1 (5)`
- Index 2: `1 -> 2 (20)`

Step 3: Lookup (Binary Search) To find edges for Node 0:

`searchsorted(Sources, 0)` → Index 0

`searchsorted(Sources, 1)` → Index 2

Slice `[0:2]` → We get both edges for Node 0! layout: section

BFS 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.

CONNECTION TO DDCM

- **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).
- **Difference:**
 - **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).

CONNECTION TO DDCM

- **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.

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`.*

Backward Induction Details

The Mathematical Engine.

The Bellman Equation

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

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

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

THE NUMERICAL PROBLEM

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

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

Scatter LogSumExp

Our GPU Solution for Numerical Stability.

THE "MAX TRICK"

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

GPU IMPLEMENTATION (`BACKWARD_INDUCTION_TENSOR.PY`)

Scatter Max: Find max Q for each source state.

```
max_val = scatter_reduce(src, index, reduce="max")
```

Stable Exponentials: Compute relative to max.

```
exp_diff = torch.exp(src - max_val[index])
```

Scatter Sum: Sum the stable exponentials.

```
sum_exp.index_add_(0, index, exp_diff)
```

Final Value:

```
result = max_val + torch.log(sum_exp)
```

Thank You

High-Performance DDCM Implementation

Powered by PyTorch & CUDA