Solving the Curse of Dimensionality in DDCM

Graph-Based Computation for Activity-Based Travel Demand Models

January 2026

Press Space for next page →

1. The Problem

Why Computation Matters

Travel Demand Models: A Brief History

1970s

Trip-Based

Aggregate zone-to-zone flows

1990s

Tour-Based

Linked trips, home-based tours

2000s+

Activity-Based

Full daily patterns, individual-level

THE KEY INSIGHT

Travel is a **derived demand**. People don't travel for its own sake—they travel to participate in *activities*. Modeling activity choice gives us a complete picture of **when, where, why, and how** people travel.

The Modeling Challenge

THE COMPLEXITY PROBLEM

"Depending on the spatial and temporal resolution considered, the number of travel patterns available could easily exceed the number of atoms in the universe."

— Västberg (2018), PhD Thesis

Two Schools of Thought

COMPUTATIONAL PROCESS MODELS

People use heuristics, not optimization.

Examples: STARCHILD, ALBATROSS, ADAPTS

✓ Can model bounded rationality

× No consumer surplus → cannot do welfare analysis

× Rules often arbitrary

RANDOM UTILITY MODELS (MEV)

People act as if they maximize utility.

Examples: MNL, Nested Logit, Mixed Logit

✓ Microeconomic foundation → welfare economics

✓ Log-sum accessibility measures

× Weak treatment of time in practice

DDCM: A Dynamic Discrete Choice Model

Västberg, Karlström, Jonsson, Sundberg (2020) — *Transportation Science*

DDCM models daily activity–travel as a **Markov Decision Process (MDP)**:

- Individuals make **sequential decisions** throughout the day
- At each moment, they consider the **expected future utility** of their actions
- Choices of destination, mode, activity, and timing are **interdependent**

Case Study

Workdays in Stockholm, Sweden.

1,240 zones × 4 modes × 6 activities

Key Properties

MEV-based → **consumer surplus** via log-sum.

Enables **cost-benefit analysis** and **accessibility measures**.

Activity-Travel Planning as an MDP

An **MDP** (S, C, q, u) models sequential decision-making under uncertainty, where S is the state space, C is set of actions (a), q is the transition probability, and u is the utility function.

STATE S_K (WHERE AM I? WHAT HAVE I DONE?)

`t`	Time of day
`l`	Location (1,240 zones)
`p`	Current activity purpose
`τ`	Duration in activity
`m`	Mode (car/PT/walk/bike)
`h`	Mandatory activity history

ACTION A_K (WHAT DO I DO NEXT?)

`d̃`	Destination (1,240 zones)
`m̃`	Mode of transport
`p̃`	Activity purpose

`mstay` = stay and continue current activity.

The Utility Function

The one-stage utility $u(s_k, a_k)$ captures preferences for state-action pairs.

$$u(s,a) = u_{\text{travel}}(t, l, m, \tilde{d}, \tilde{m}) + u_{\text{start}}(l, t, h, \tilde{p}) + u_{\text{act}}(t, p, \tau) + \varepsilon(a)$$

u_{travel}

Disutility of travel based on travel time, cost, and mode-specific constants. Captures the "pain" of traveling.

u_{start}

When and where to start an activity. Includes time-of-day preferences and location attractiveness.

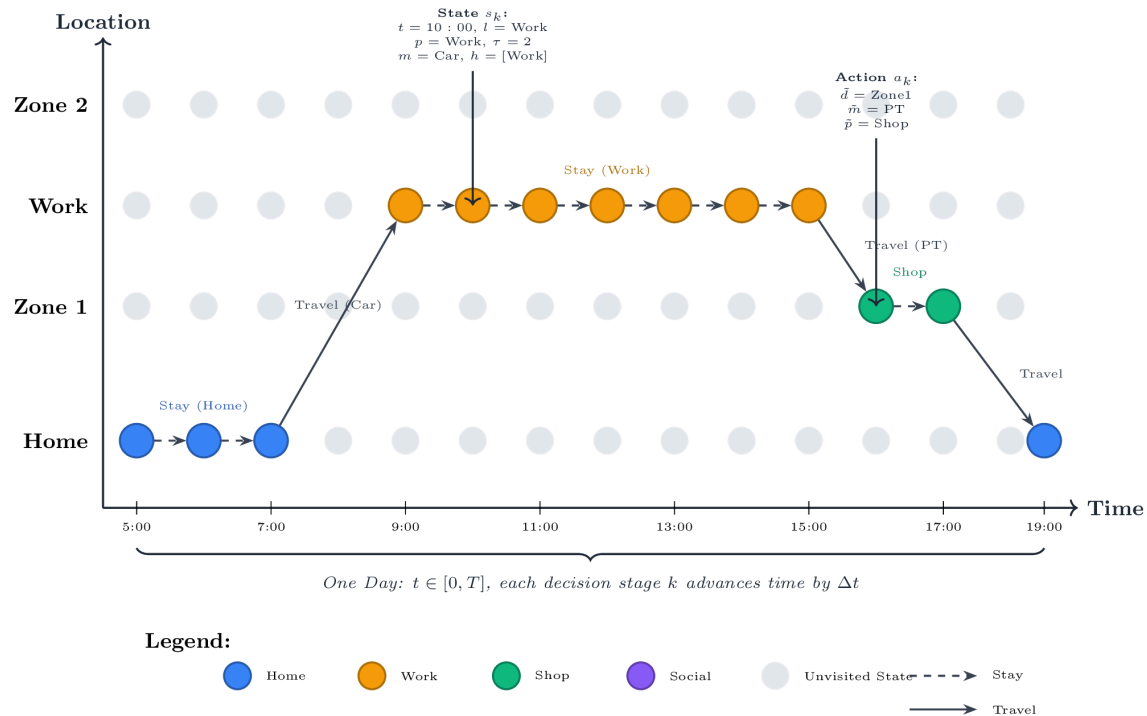
u_{act}

Marginal utility of spending time in an activity. Can depend on duration already spent.

$\varepsilon(a)$

Random utility term (i.i.d. Gumbel distributed) → Logit choice probabilities.

Visualizing a Daily Activity Pattern



A typical workday: Morning at Home → Travel to Work → Stay at Work → Travel to Shopping → Travel Home

The Bellman Equation

The value function can be defined recursively (Bellman 1957, Rust 1987)

PRINCIPLE OF OPTIMALITY

At each state, the agent chooses action *a* that maximizes:

```
Immediate utility + Expected future utility
```

RANDOM UTILITY

Add i.i.d. Gumbel error term ε(a):

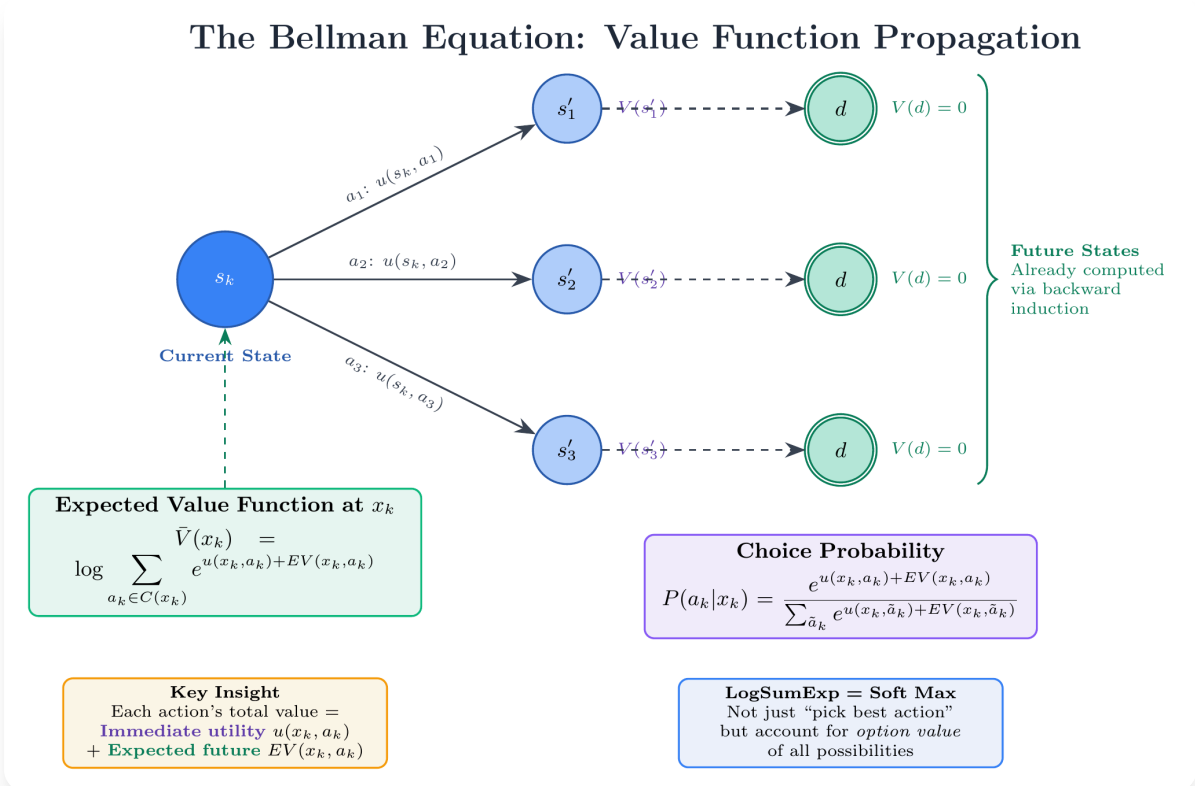
Total utility includes randomness

→ Leads to **Logit** choice probabilities

THE EXPECTED VALUE FUNCTION (SOLVED RECURSIVELY)

$$\bar{V}(x_k) = \log \sum_{a_k \in C(x_k)} \exp\left[u(x_k, a_k) + EV(x_k, a_k)\right]$$

The Log-Sum: Expected Maximum Utility



Why Log-Sum-Exp Matters

WHY LOG-SUM-EXP?

- The **Gumbel distribution** gives a closed-form for "expected maximum"
- Analytically **differentiable** — enables gradient-based estimation

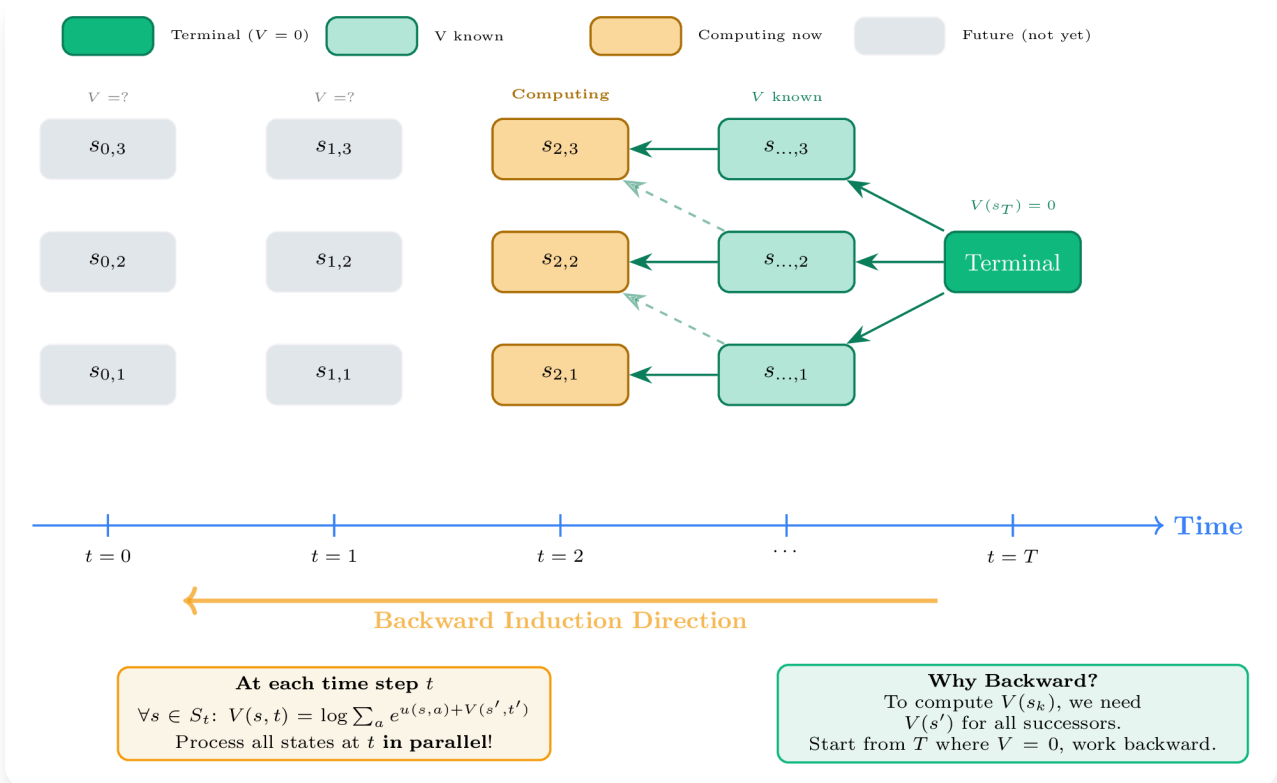
ECONOMIC INTERPRETATION

- **V(s)** = Expected Maximum Utility
- = **Consumer Surplus** (welfare economics)
- = **Accessibility** measure
- = Willingness to pay for options

This is why DDCM maintains MEV (Multivariate Extreme Value) structure

→ **Enables proper welfare economics and cost-benefit analysis**

Solving: Backward Induction



Backward Induction: The Algorithm

Step 1

Set terminal condition

```
V(s_T) = 0
```

for valid end states

Step 2

Work backward in time

```
t = T-1, ..., 0
```

compute V(s,t) for all states

Key Benefit

Future V is always known

when needed!

No iterative solving

Complexity: O(|S| × |A|) per time step — linear in graph size, not cubic like matrix inversion

Why DDCM? Bridging the Gap

> *"Models which succeed in [treating interdependent travel patterns] are either* ***not consistent with microeconomic theory*** *or are* ***prohibitively time consuming to evaluate****."*
>
> — Västberg (2018), PhD Thesis

DDCM COMBINES THE BEST OF BOTH APPROACHES:

Maintains MEV structure → Consumer surplus via log-sum (economics ✓)

Consistent time treatment → Time-space constraints naturally handled

Sequential decisions → Interdependence captured implicitly (realism ✓)

Key insight: Instead of "which travel pattern?"

The Remaining Problem: Computation

DDCM IS THEORETICALLY SOUND, BUT...

- **~10 seconds per agent** with 1,240 zones
- Full city (100K agents) → **1,000+ CPU-days**
- Required Swedish National Supercomputer

WHAT WE WANTED TO SOLVE

- Make within-day DDCM **computationally tractable**
- Enable **population-scale simulation** on consumer hardware
- Preserve all theoretical properties (MEV, log-sum, time consistency)

The Curse of Dimensionality

Dynamic Discrete Choice Models model sequential decision-making over time.

9 DIMENSIONS OF COMPLEXITY

- **Location**: ~1240 zones
- **Mode**: 4-8 types
- **Activity**: 10 types
- **Time**: 96 steps (15-min)
- **History**: Mandatory progress

= Billions of States

COMPUTATIONAL COST

Västberg (2020): 4-10s per agent

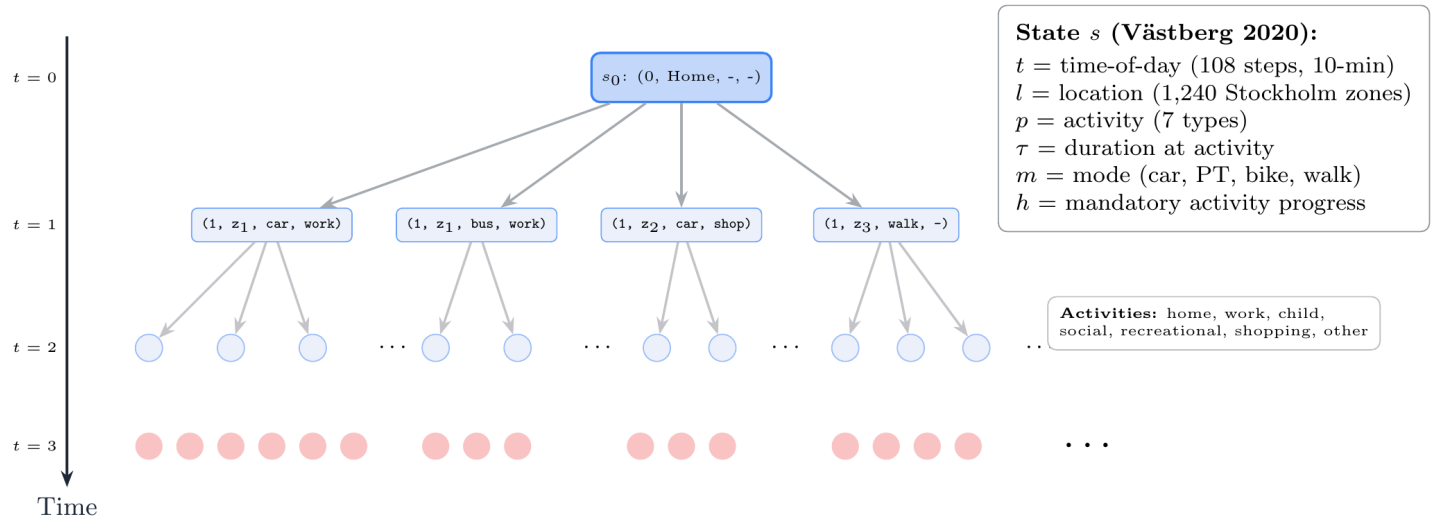
Full city: 1,000+ days

O(|States| × |Actions|) per time step

THE BOTTLENECK: STATE-ACTION ENUMERATION

Backward induction requires evaluating the value function for **every reachable state**. With ~25M states × ~30K actions = **~750 Billion operations**.

The Curse of Dimensionality



Source: Västberg et al., Transportation Science, 2020

Exponential growth of state space over time

The Question That Started Everything

"What if we could avoid computing what's impossible?"

TRADITIONAL APPROACH

- Enumerate ALL possible state combinations
- Most are never actually reachable
- Brute-force computation → exponential cost

OUR INITIAL HYPOTHESIS

- Only ~0.1% of states are reachable
- Physical and temporal constraints eliminate most
- Build only what's possible → massive savings

Research Objectives

PRIMARY OBJECTIVE

Make Dynamic Discrete Choice Models (DDCM) **computationally tractable** for population-scale activity-based travel demand modeling.

TARGET OUTCOMES

- ✓ Reduce computational burden by orders of magnitude
- ✓ Enable simulation on consumer hardware (GPU)
- ✓ Preserve theoretical properties (MEV, welfare economics)

THE CHALLENGE

Current state: ~10 seconds per agent → 100,000 agents = **1,000+ CPU-days**

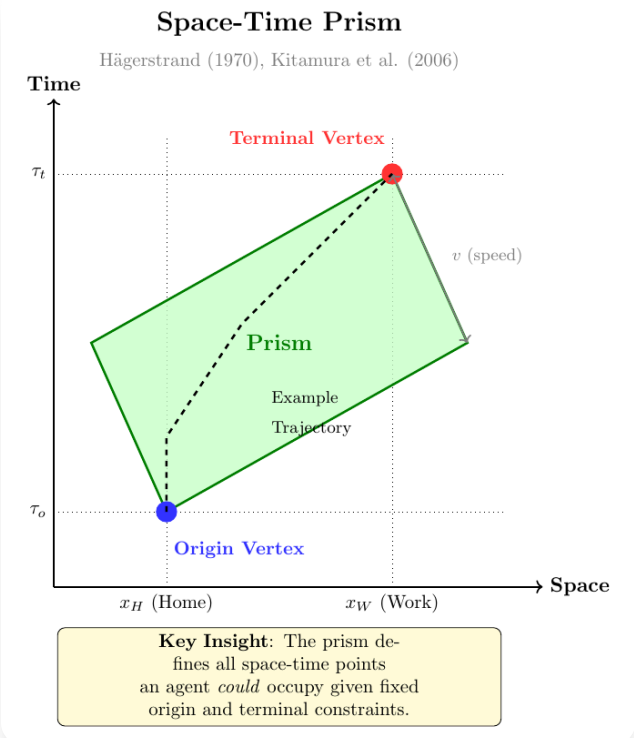
Goal: Make this tractable while maintaining **optimality** and **economic interpretability**

2. Theoretical Foundations

The Reachability Concept

The Space-Time Prism

Hägerstrand (1970): Not everywhere is reachable from everywhere



KEY CONCEPT

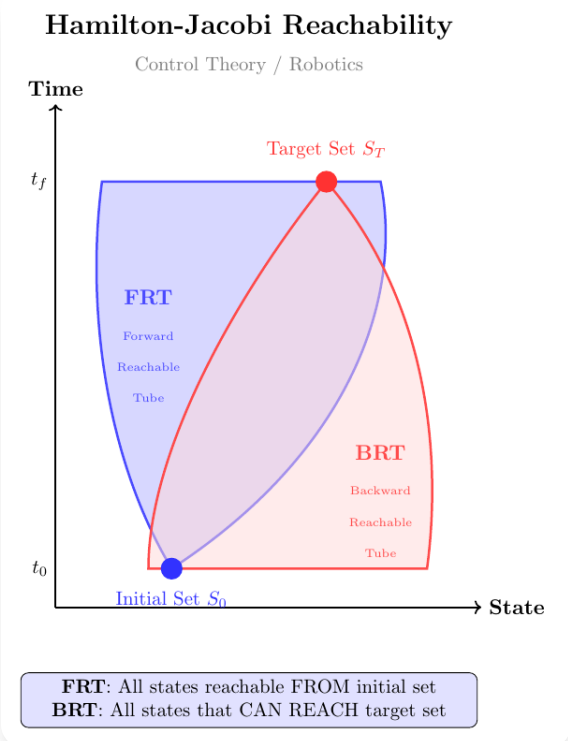
- **Origin Vertex**: (Home, τ_o) — when you can leave
- **Terminal Vertex**: (Work, τ_t) — when you must arrive
- **Speed Constraint**: Limits how far you can travel
- **The Prism**: All space-time points you *could* occupy

Activity scheduling has inherent space-time constraints!

See also: Oyama & Hato (2019) - Prism-based path set restriction for MTA (Markovian Traffic Assignment)

Hamilton-Jacobi Reachability

"We adapt ideas from control theory to discrete choice modeling"



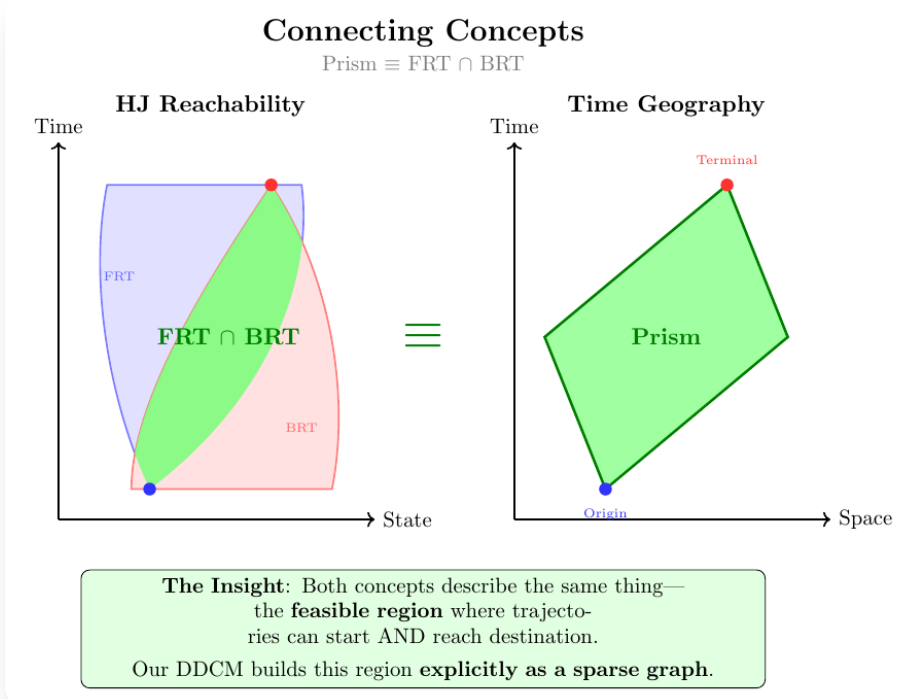
TWO REACHABLE TUBES

- FRT (Forward): All states reachable FROM start
- BRT (Backward): All states that CAN REACH target
- Intersection: Compute value function only here!

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

Reference: Bansal et al. (2017) - Hamilton-Jacobi Reachability: A Brief Overview and Recent Advances

Connecting the Concepts



Viability Kernels

DEFINITION

The set of states from which there exists *at least one* trajectory that stays within constraints indefinitely (or until reaching the target).

If you leave the kernel, you violate constraints (e.g., miss your mandatory activity window).

CONNECTION TO OUR APPROACH

The viability kernel concept justifies why we only compute on **Forward Reachable Tube (FRT) ∩ Backward Reachable Tube (BRT)**.

Reference: Coquelin et al. (2007) - A Dynamic Programming Approach to Viability Problems

Research Questions (Part 1)

BASED ON REACHABILITY THEORY

RQ1: State Space Reduction

Can reachability analysis (FRT ∩ BRT) effectively reduce the state space explosion in DDCM?

Expected: 99%+ reduction from viability kernel pruning

RQ2: GPU Parallelization

What speedup is achievable through GPU-accelerated graph traversal and backward induction?

Target: Memory bandwidth-limited performance (19× theoretical)

Next: We'll address these through our three-phase computational pipeline

3. Building the Solution

Key Technical Discoveries

Discovery 1: DP as Graph Problems

OLD: MATRIX INVERSION

$Z = (I - M)^{-1} b$

- Even if M is sparse, inverse is dense
- $O(N^3)$ complexity
- 8.7 TB for 1.5M states

NEW: GRAPH TRAVERSAL

`BFS / Backward Induction`

- Graphs are naturally sparse
- O(V + E) complexity
- 1.3 GB using CSR format

THE KEY INSIGHT

Dynamic Programming → Graph Traversal → Sparse Problems → Tensor Operations

Why DP Can Be Viewed as Graph Traversal?

BELLMAN EQUATION STRUCTURE

$V(s) = \max_a \{ u(s,a) + V(s') \}$

- **States (s)** → Graph Vertices
- **Actions (a)** → Graph Edges
- **Transitions (s → s')** → Edge Connections

GRAPH REPRESENTATION

Each state is a node, each possible action is an edge

- **Forward Pass** = BFS (discover reachable states)
- **Backward Induction** = Reverse topological traversal
- **Finite horizon** = Directed Acyclic Graph (DAG)

Key Insight: Backward induction in finite-horizon MDPs is mathematically equivalent to dynamic programming on a DAG

Reference: Dudzik et al. (2022) - "Graph Neural Networks are Dynamic Programmers"

Why Tensor Representation?

OBJECT-BASED (PYTHON)

```
class State:
  time: int
  zone: int
  activity: int
  ...  # 8 fields
```

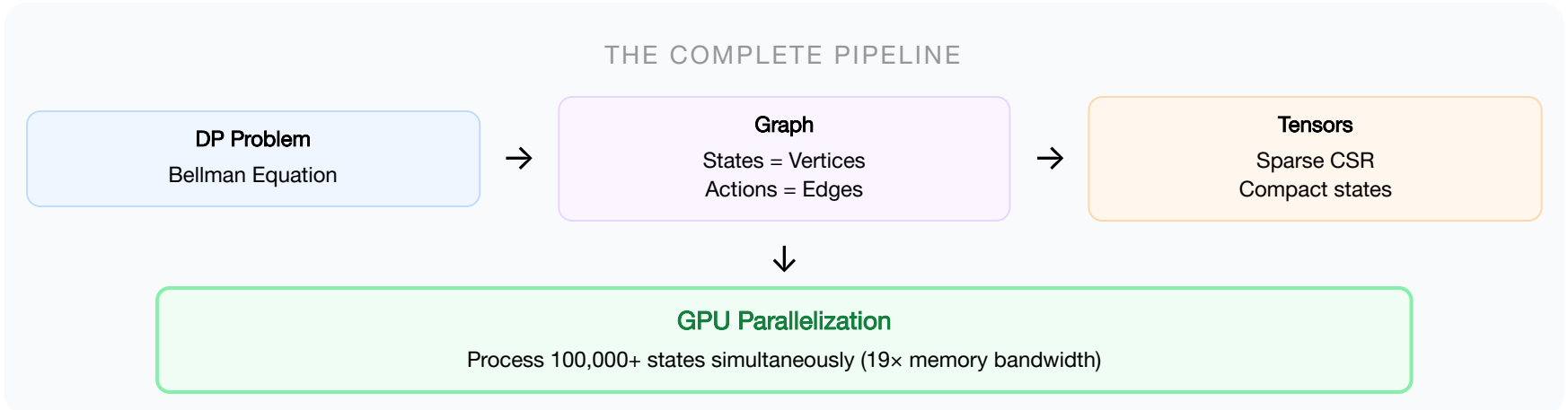
- Object overhead: ~350 bytes/state
- 1.5M states = 525 MB (metadata only!)
- Python loops: ~10,000 states/sec
- ❌ Cannot leverage GPU parallelism

TENSOR-BASED (PYTORCH)

```
states = torch.tensor([
  [time, zone, act, ...],
  ...  # Shape: (N, 8)
], dtype=torch.int32)
```

- Compact: 32 bytes/state (8 × 4 bytes)
- 1.5M states = 48 MB (11× reduction!)
- GPU operations: 100,000+ states/sec
- ✓ Full GPU parallelization

Putting It All Together



1. DP → Graph

Bellman equation becomes graph traversal (BFS + Backward Induction on DAG)

2. Graph → Tensors

Sparse CSR graph + tensor states fit in GPU memory (1.3 GB + 48 MB)

3. Tensors → GPU

Vectorized operations on 100,000+ states simultaneously

Discovery 2: GPU Parallelization

Level-Synchronous BFS: Process all states at each time step in parallel

CSR: COMPRESSED SPARSE ROW

Dense Matrix	8.7 TB ✗
Edge List (COO)	2.6 GB
CSR	1.3 GB ✓

O(1) edge lookup vs O(log E)

MEMORY BANDWIDTH

CPU (DDR4)	25 GB/s
GPU (GTX 1080 Ti)	484 GB/s

19× bandwidth advantage!

Discovery 3: The Three-Phase Pipeline

1. Forward Reachability

Build sparse graph via BFS

~30s

2. Backward Induction

Compute V(s,t) on graph

~2s/home

3. Simulation

Sample trajectories

~0.4s

Phase 1: Forward Pass Implementation

Goal: Discover reachable states via **Level-Synchronous BFS**.

THE ALGORITHM

- **Time Buckets**: Process states layer-by-layer (t=0, 15, 30...)
- **Deduplication**: `torch.unique` merges identical paths at each step
- **Parallel Expansion**: GPU kernel generates next states for 100k+ states at once

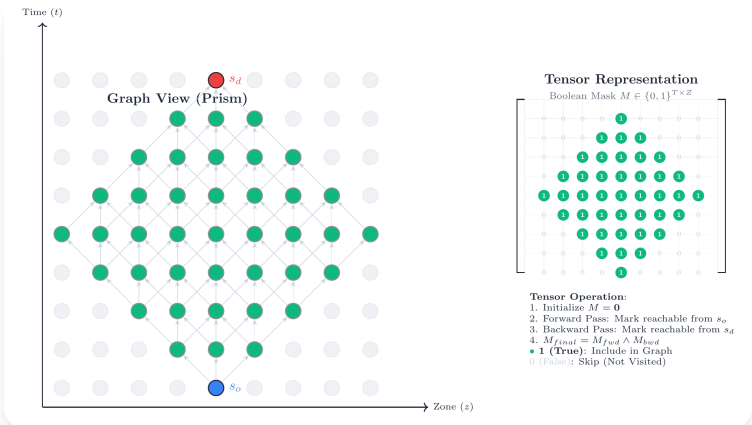
WHY IT WORKS

Instead of processing one agent at a time (DFS), we process the **entire population's potential states** as a wavefront.

`File: planning/forward_pass_tensor.py`

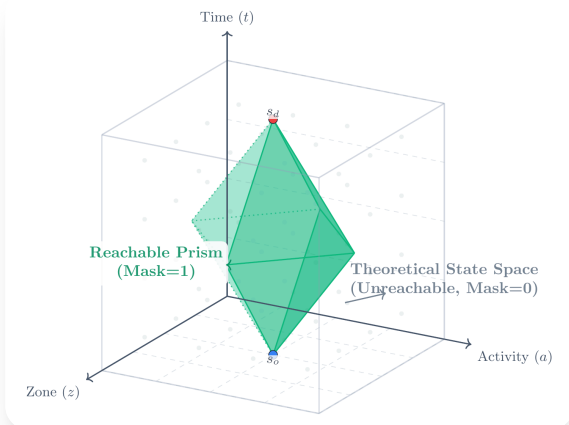
Visualizing Phase 1: The Space-Time Prism

2D GRID & TENSOR MASK



Graph View (Left) matches Tensor Mask (Right).

3D REACHABLE VOLUME



Green Prism (Reachable) inside Gray Cube (Theoretical Space).

Phase 2: Graph Construction Implementation

Goal: Convert raw states into a **CSR (Compressed Sparse Row)** Graph.

CSR FORMAT EFFICIENCY

Memory: 50% savings vs Edge List.

Speed: O(1) edge lookup for Backward Induction.

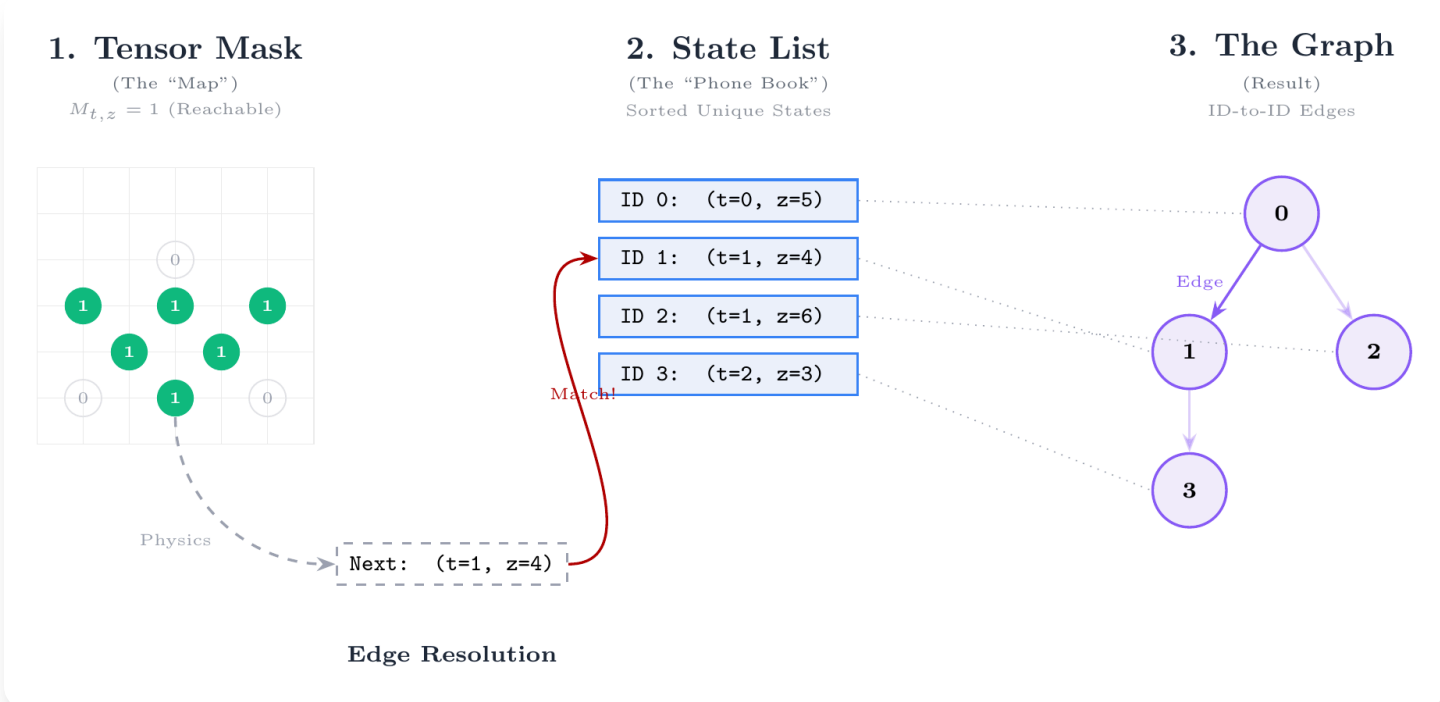
```
row_ptr: [0, 3, 5, ...]
→ Node 0: edges at [0:3], Node 1: edges at [3:5]
col_idx: [1, 2, 5, 3, 4, ...]
→ Node 0 → {1, 2, 5}, Node 1 → {3, 4}
```

THE PIPELINE

1. **Hash**: 8 columns → 64-bit Int
2. **Sort**: Create "Phone Book" of states
3. **Map**: Resolve edges to Global IDs

`File: planning/graph_builder_tensor.py`

Visualizing Phase 2: From Tensors to Graph



Phase 3: Backward Induction Implementation

Goal: Solve the Bellman Equation for $V(s,t)$ in parallel.

$$V(s,t) = \log \sum_{a} \exp\left[u(s,a,t) + \beta V(s',t')\right]$$

VECTORIZED SOLVER

We process **all states at time t** simultaneously.

Scatter-Reduce: Aggregates 326M edges in ~2 seconds.

NUMERICAL STABILITY

Using the **LogSumExp "Max Trick"** to prevent overflow:

```
log(sum(exp(x))) = m + log(sum(exp(x-m)))
```

File: planning/backward_induction_tensor.py

4. The Challenge

Individual Heterogeneity

Research Question (Part 2)

A NEW CHALLENGE EMERGES

RQ3: Individual Heterogeneity

How can we handle individual heterogeneity (different home/work locations) without exponential graph replication?

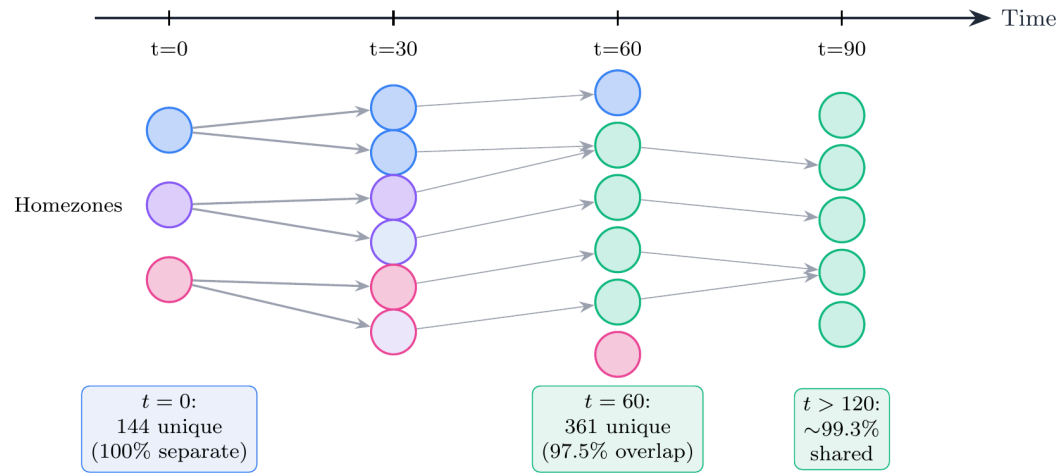
Hypothesis: Abstract zone roles via RMDP, value function calculation via approximate approach

The Problem

- 144 home × 144 work = **20,736 graphs**
- Each ~60 seconds → **316+ hours!**

Solution 1: RMDP Zone Abstraction

"One Universal Graph to rule them all"



What is RMDP?

RMDP = Relational Markov Decision Process (Boutilier et al., 2001)

The Problem with Standard MDPs

You must explicitly list every state.

Different (Home, Work) pairs = different MDPs.

144 × 144 = 20,736 unique MDPs!

The RMDP Solution

Use **abstract roles** (HOME, WORK) instead of concrete zones.

Decision structure is **identical** regardless of binding!

20,736 → 1 graph

Scope: Abstracts zone locations (HOME, WORK). Different mandatory activity sequences still require separate graphs.

RMDP: Core Insight & Performance

"One Universal Graph to rule them all"

THE CORE INSIGHT

- **Abstract State**: Use roles like `HOME`, `WORK`
- **Universal Graph**: Built once using abstract roles
- **ZoneBinding**: Resolves to concrete zones at runtime

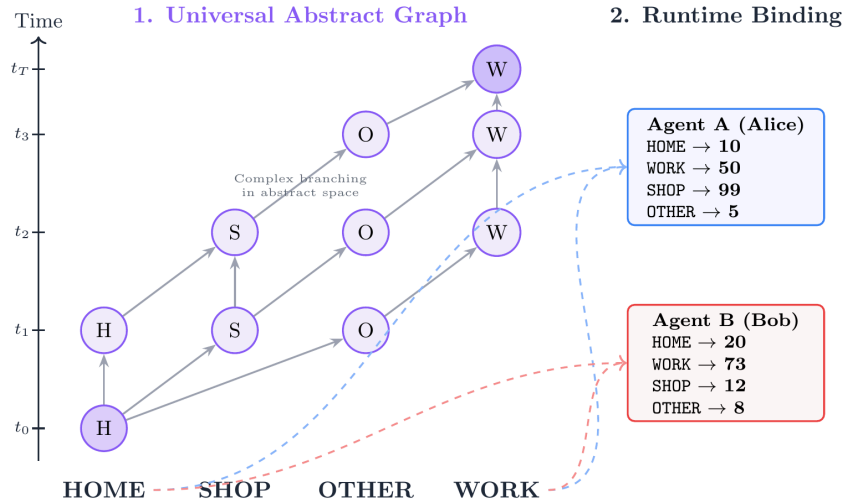
PERFORMANCE

Graphs needed	20,736 → 1
Total time	316h → 47s
Speedup	~24,000×

Key: Decision structure is identical regardless of zone binding → Build once, use everywhere!

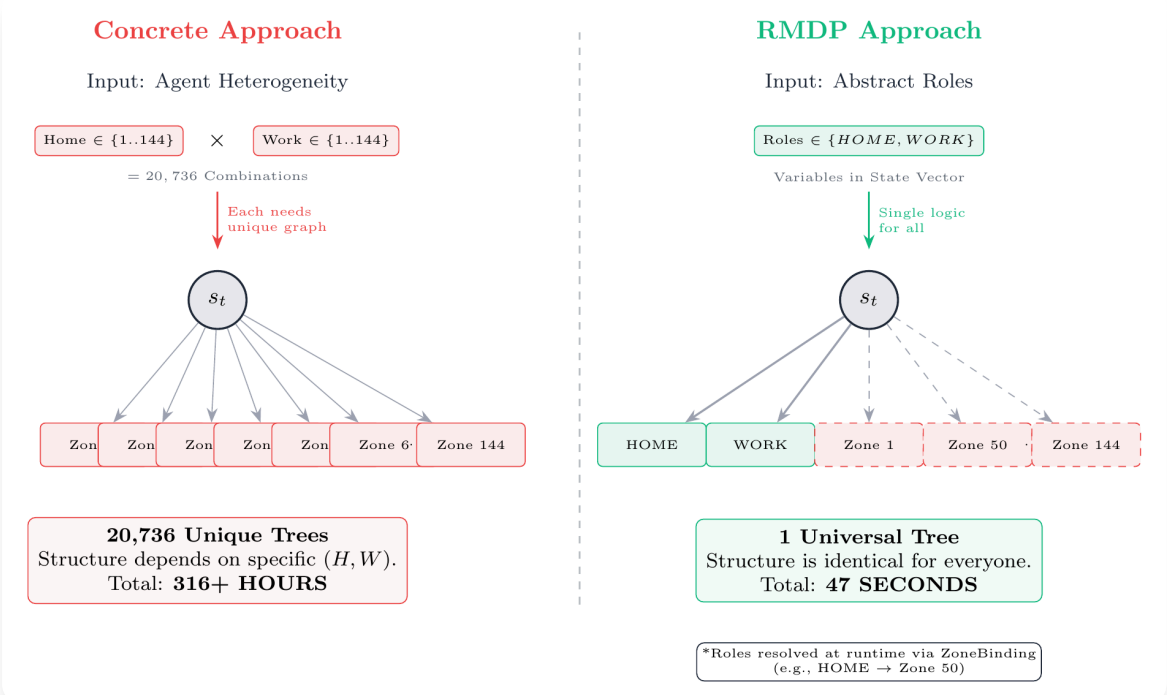
Zone Binding: The Concept

"One Graph, Many Agents"



Same Graph, Different Reality:
When the graph says "Go to WORK", Alice goes to Zone 50, Bob goes to Zone 73.
This allows **millions of agents** to share a single graph structure in GPU memory.

Graph Explosion: Visualization



Solution 2: Basis Function Approximation

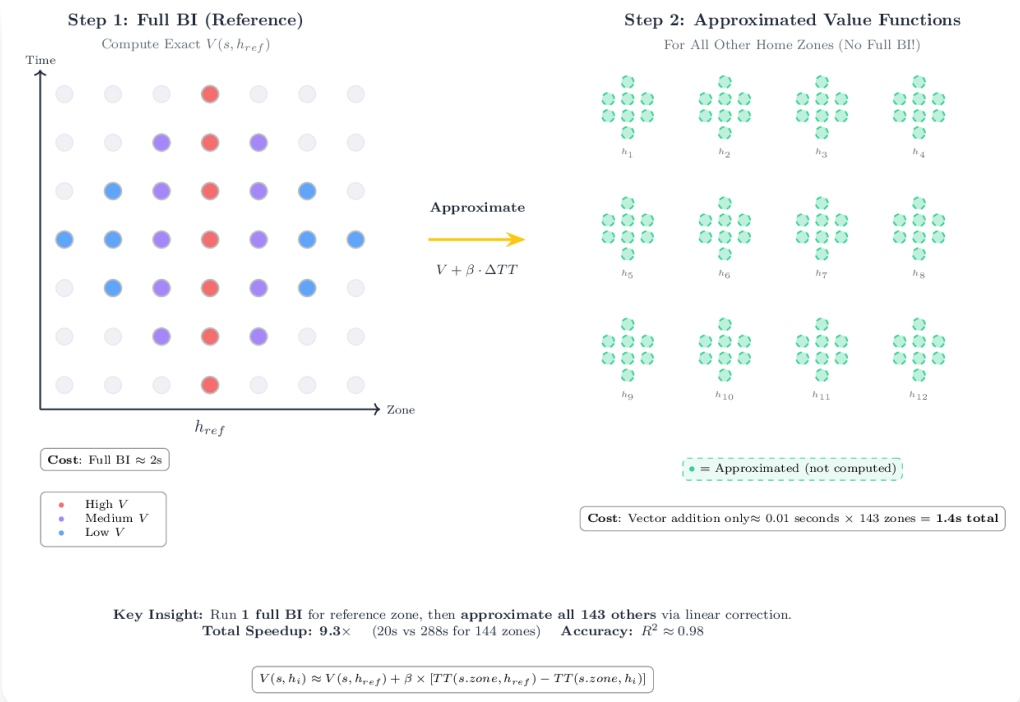
Problem: Still need separate backward induction per home zone

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

METHODOLOGY

- Train on 8 representative zones
- Interpolate for remaining 136
- **Accuracy: $R^2 = 0.982$**
- **Speedup: 9.3×**

Basis Function: Visualization



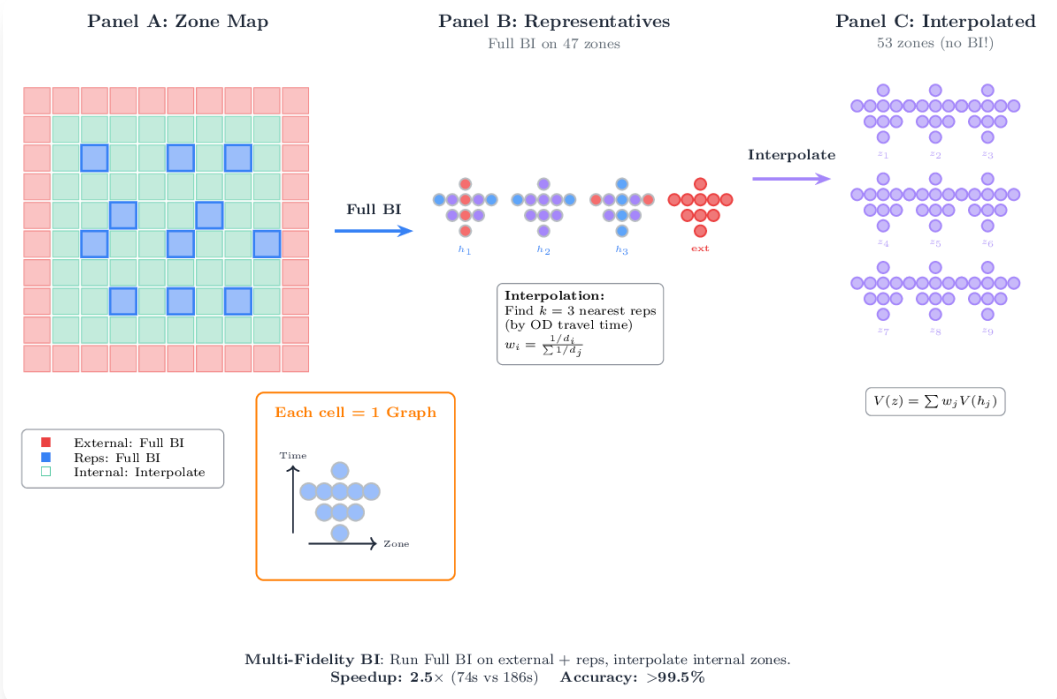
Alternative: Multi-Fidelity BI

Idea: Use high-fidelity (exact) BI for some zones, interpolate for others. Same approach like IDW (Inverse Distance Weighting) in spatial interpolation.

PERFORMANCE

- Full BI for 15 representatives
- Interpolate the rest
- **Accuracy: 99.6%** (Higher than Basis Function)
- **Speedup: 2.5×** (Lower than Basis Function)

Multi-Fidelity: Visualization

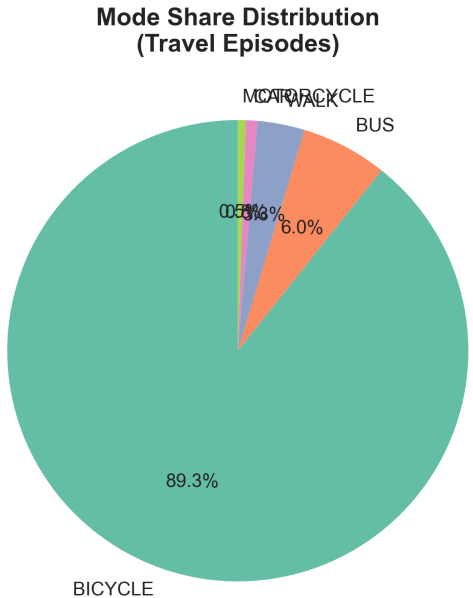


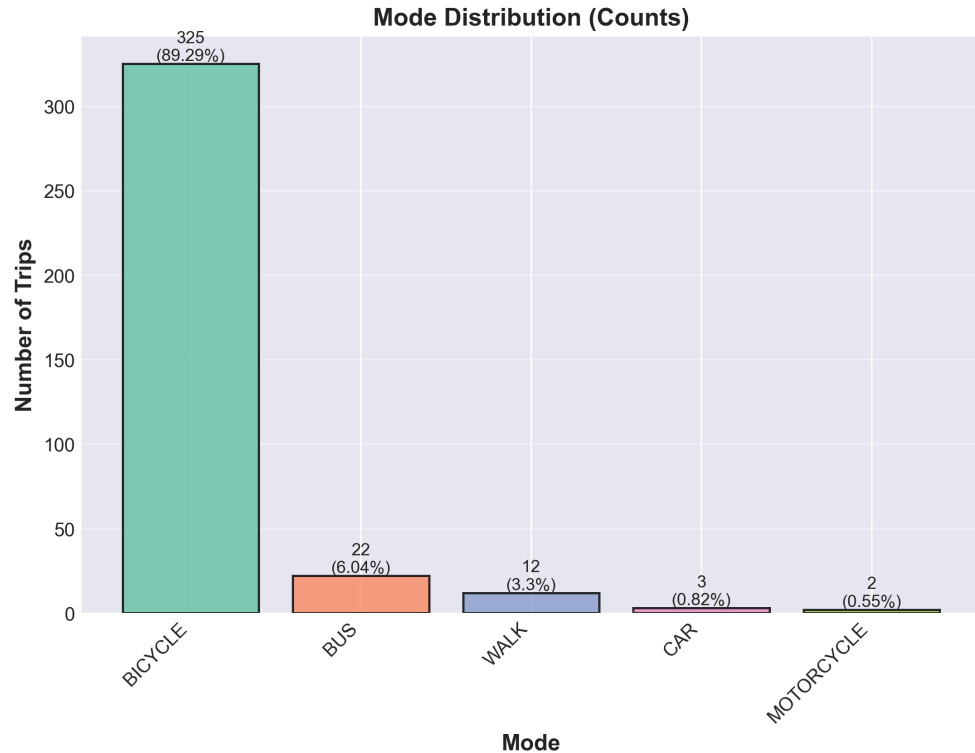
Performance Comparison

Approach	Time (1 Agent)	Key Speedup
No Pruning (Baseline)	**71.1 hours** ⚠️	1×
Dict-Based CPU	108s	2,370×
Tensor-Based CPU	65s	3,940×
GPU CUDA	8s	**32,000×**
Universal Graph	7s	**24.7×** vs per-home
Basis Function BI	5s	**9.3×** vs full BI
RMDP	8.7s	**29,382×**

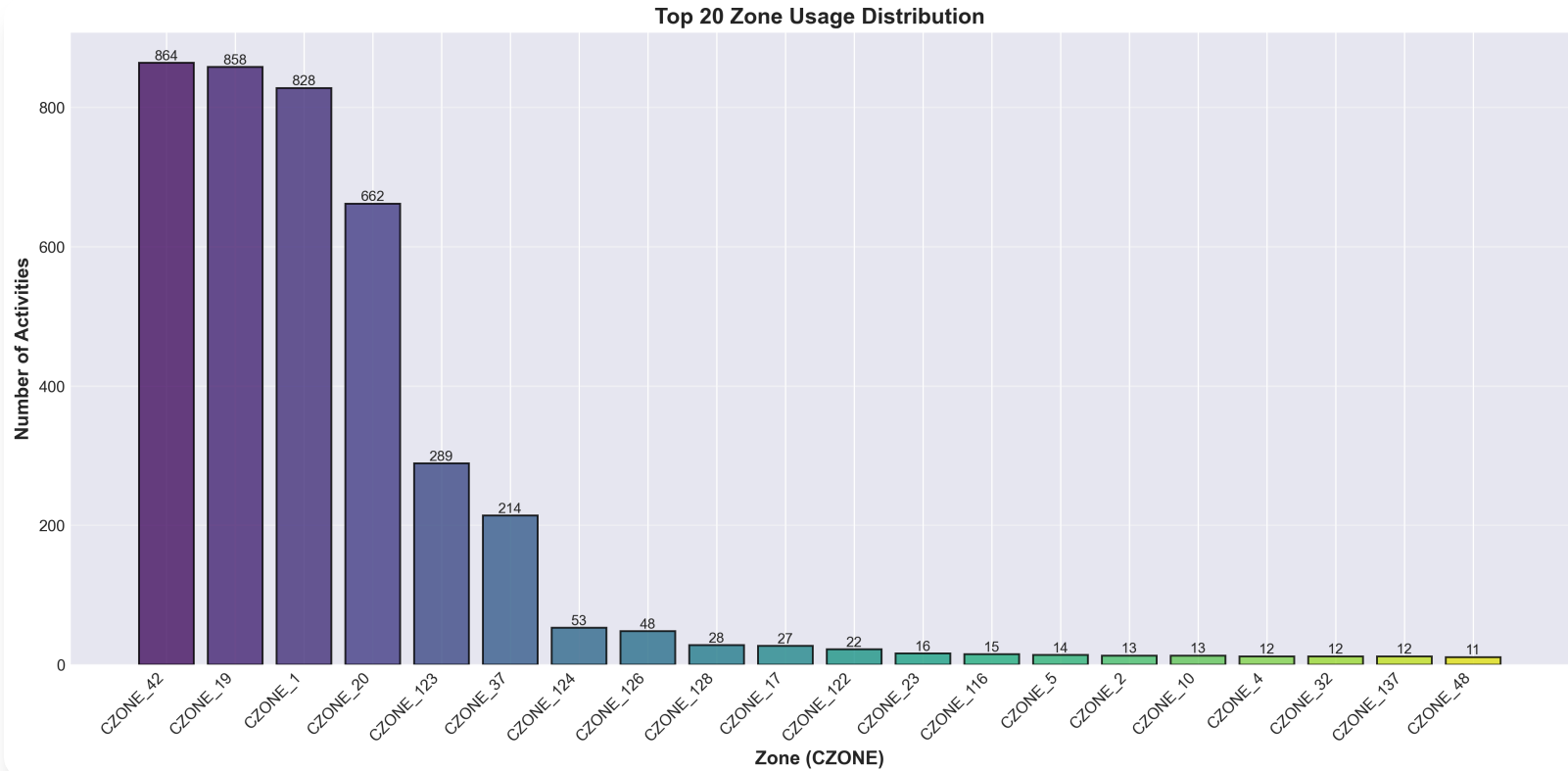
Simulation Results

Result: Mode Distribution

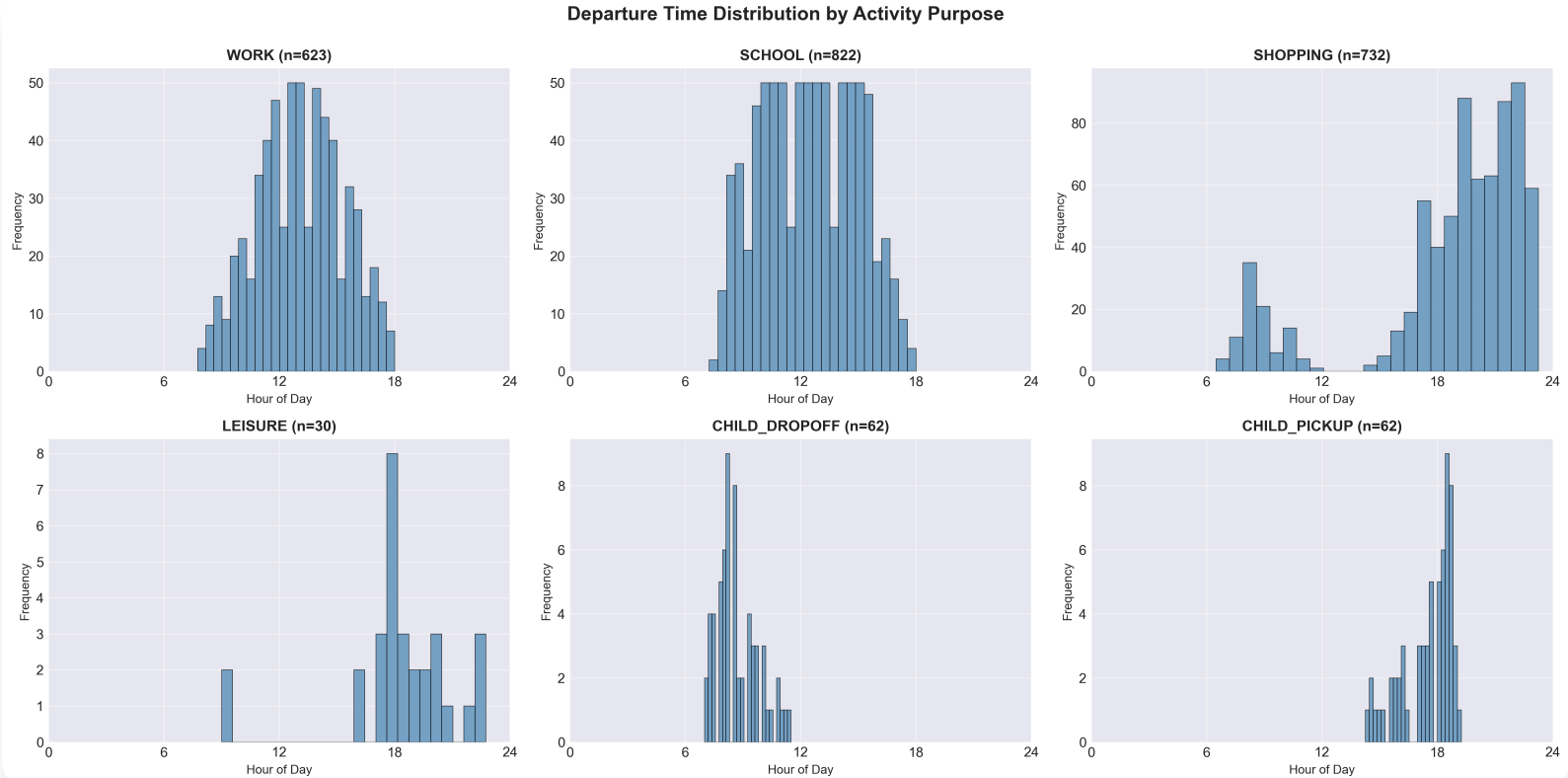




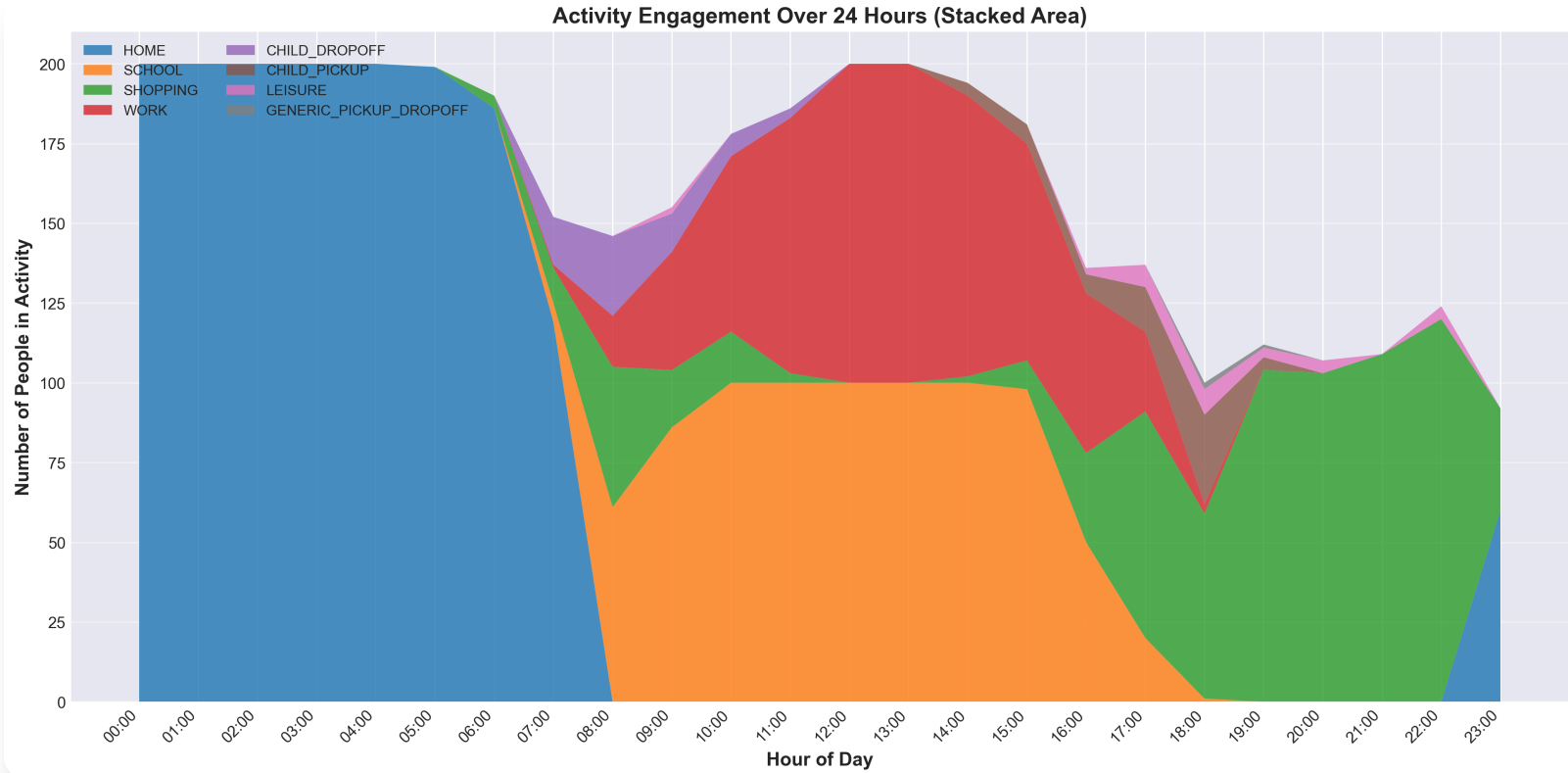
Result: Zone Usage Distribution



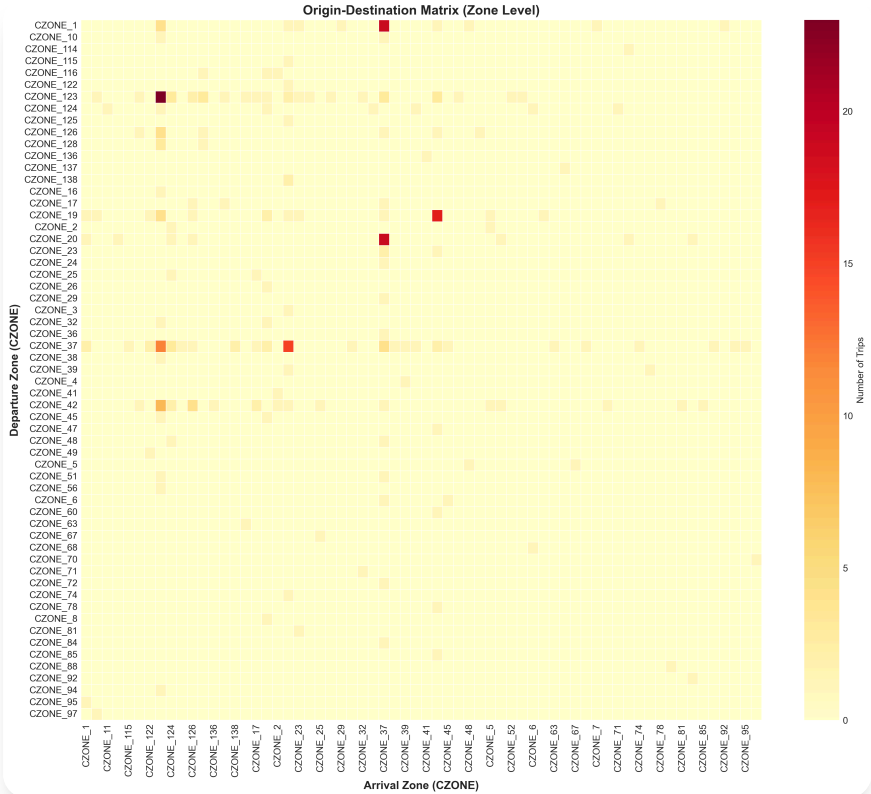
Result: Departure Time by Purpose



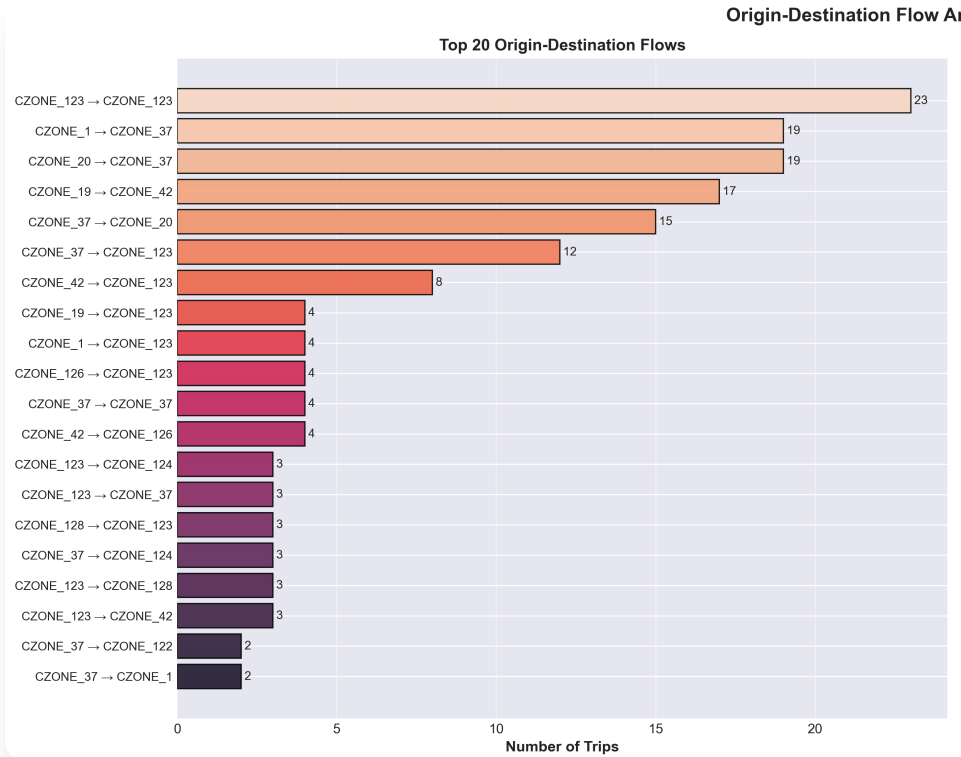
Result: Activity by Time (24hr)



Result: OD Matrix (Czone Level)



Result: Top OD Flows




```
OD Flow Statistics:

Total OD pairs: 30
Total trips: 175
Unique origins: 59
Unique destinations: 64

Top 10 pairs: 125 trips (71.4%)
Top 20 pairs: 155 trips (88.6%)

Most frequent flow:
CZONE_123 → CZONE_123
(23 trips)
```

5. Future Work

The Road Ahead

Västberg's Identified Gaps

From the PhD thesis (2018), several challenges remained unresolved:

1. Estimation

NFXP too slow for repeated model evaluation.

2. Correlation

i.i.d. error terms too restrictive.

3. Simulation

~10s per agent, millions needed for traffic assignment.

4. Household Interaction

Joint activities require multi-agent coordination.

5. Long-Term Decisions

Car ownership, work/home location as exogenous.

6. Traffic Assignment

MATSim integration for demand forecasting.

This Work: Tackling Simulation

WE FOCUSED ON GAP #3: SIMULATION

Making DDCM computationally tractable for population-scale simulation.

Problem

- ~10 seconds per agent
- 100K agents = 1,000+ CPU-days
- Required Swedish National Supercomputer

Our Solution

- GPU acceleration + reachability pruning
- RMDP zone abstraction
- **8.7 seconds for ALL configurations**

29,382× speedup → Simulation feasible → Estimation, MATSim now tractable

The Insight: Reachability as Foundation

While solving the simulation problem, we discovered that **forward reachability analysis** is the key computational insight.

WHAT REACHABILITY GIVES US

- **State space pruning**: Only compute feasible states (0.1% of theoretical)
- **Time-space constraints**: Naturally encoded in the reachable tube
- **Individual heterogeneity**: Different constraints → different reachable sets

Key Realization

Reachability analysis isn't just an optimization—it's a **general framework** for handling any constraint that can be expressed as "what states are reachable?"

From Specific Solution to Meta-Framework

RQ4: Generalizability

Can the reachability-based approach generalize to other constraints?

Answer: Yes, through the Conditions Framework

Current Work (Specific)

- Hard-coded time-space constraints
- Fixed activity requirements
- Single-agent focus

Conditions Framework (General)

- **Declarative constraint specification**
- Composable condition algebra
- Multi-agent reachability

The Generalization: Define constraints as composable "conditions" that modify the reachable set.

Conditions Framework: Applications

Household Equity

Track travel burden *within* households.

Gender equity under congestion pricing.

Joint Activities

Unified abstraction for care activities.

Childcare, eldercare, shared meals.

Work Flexibility

Evaluate policy impacts.

4-day week, flextime, remote work.

```
Policy Questions → Condition Objects → Reachability Engine → Results
```

Same computational foundation (this work) powers all applications.

Summary: Research Roadmap

Gap	Status	Enabled By
Simulation	✓ Solved (29,382×)	This work
Estimation	Ready to implement	Speedup
Traffic Assignment	Possible to integrate	Speedup
Correlation	Research needed	-
Households	Conditions Framework	Reachability
Long-term	Conditions Framework	Reachability

Research Contributions

1. INTEGRATION GAP

Discrete Choice + Reachability Analysis

2. PERFORMANCE

29,382× speedup via RMDP

3. THEORETICAL BRIDGE

Space-Time Prism ≡ FRT ∩ BRT

4. FRAMEWORK

Condition-based MDP pattern

Thank You

Solving the Curse of Dimensionality in DDCM

Graph-Based Computation | PyTorch & CUDA | CSR Graphs | Level-Sync BFS

29,382× speedup | 71 hours → 9 seconds | Optimal solutions preserved

Appendix

Appendix: Utility Function Overview

Total Utility = Travel Utility + Activity Utility + Demographic Effects

Travel Utility

$U = \beta_{mode} + \beta_{time} \times TT + \beta_{cost} \times TC$

Mode constant + time disutility + cost

Activity Utility

$U = \beta_{activity} + \beta_{size} \times \log(zone\_attr)$

Activity constant + zone attractiveness

Timing Preferences

Gaussian peak around optimal time

Work: 8:40AM, School: 8:06AM

Appendix: Transport Mode Parameters

Mode	Mode Constant	Time Coef (per min)	Cost Coef
Car	-1.5	-0.084	-0.05
Train	-3.8	-0.038	-0.05
Walk	-1.5	-0.060	—
Bus	-3.8	-0.038	-0.05
Bicycle	-1.5	-0.055	—

Value of Time (Car):
VOT = $|\beta_{time} / \beta_{cost}|$ = 0.084/0.05 = **1.68**
→ Willing to pay $1.68 to save 1 minute

Same-Zone Walk Penalty: -0.53
Prevents unrealistic intra-zonal walks

Appendix: Zone Attractiveness

Discretionary activities use zone attributes for location choice:

Shopping

Base: -2.7, Size Coef: 0.60

Attributes: mass retailers, food retail, clothing, daily goods

$$U = -2.7 + 0.6 \times \log(\Sigma\ \text{retail\_count})$$

Leisure

Base: -1.5, Size Coef: 0.50

Attributes: entertainment, sports, restaurants

$$U = -1.5 + 0.5 \times \log(\Sigma\ \text{leisure\_count})$$

Appendix: Mandatory Activity Timing

Soft timing preferences (Västberg et al. 2020): Linear interpolation between breakpoints

Work Timing

7:00 AM	-5.0
8:00 AM	-1.0
8:40 AM	0.0 (ref)
10:00 AM	-3.0
12:00 PM	-15.0

School Timing

7:00 AM	-0.5
7:30 AM	-0.3
8:06 AM	0.0 (ref)
9:00 AM	-0.8
10:00 AM	-2.5

Reference times (β=0) aligned with **empirical means** from travel diary data

Appendix: Activity Change Penalties

Leaving Penalties

(Activity → TRAVEL)

Home/Work/School	-3.20
Shopping/Leisure	-3.60
Hospital	-3.60

Arrival Penalties

(TRAVEL → Activity)

All activities	-3.20

Purpose: Captures inertia/transaction cost of switching activities

Appendix: Stay-at-Home Utility

Time-varying utility for being at home:

Time	Utility/min	Interpretation
12 AM - 6 AM	High (0.4/60)	Night rest
6 AM - 10 AM	Declining	Morning departure
10 AM - 12 PM	Low (0.05/60)	Work hours
12 PM - 6 PM	Low	Daytime activities
6 PM - 12 AM	Increasing	Evening return

Creates **"U-shaped"** preference: high utility for home in early morning and late evening

Appendix: Higashihiroshima Case Study

Geographic Scope

Location	Higashihiroshima City, Hiroshima Prefecture, Japan
Zones	**13 Bzones** (aggregated from 144 Czones)
Population	~1.68 million (aggregated)
Zone 14	Excluded (external regions)

Data Sources

Car Times	LOS_car.csv (routing data)
PT Times	Jorudan routing API
Zone Attrs	Hiroshima Prefecture GIS
Survey	Higashihiroshima travel diary

Note: Zone 14 (Tottori, Shimane, Okayama, etc.) excluded due to missing car LOS data and only 0.6% of trips.

Appendix: Parameter Calibration Status

Category	Status	Notes
Mode Constants (Car, Train, Bus, Walk)	⚠️	Need calibration
Mode Constants (Bicycle, Motorcycle, Taxi)	❌	TODO - placeholder
Time Coefficients	⚠️	Reasonable estimates
Zone Attractiveness	⚠️	Need calibration
Sub-attribute Weights	❌	All 0.0 currently
Activity Duration	⚠️	Initial estimates
Timing Preferences	✅	Empirically derived
Activity Change Penalties	⚠️	Västberg-style

⚠️ Current parameters are placeholder values.