Graph-Based Human Mobility Prediction with Weighted Cell-Transition Graphs and Node2Vec Embeddings

Davide Lugli
Graph Analytics — University of Modena and Reggio Emilia
Email: 269692@studenti.unimore.it

***Abstract*—This paper addresses the problem of predicting user trajectories in a city partitioned into a grid, as proposed in the ACM SIGSPATIAL Cup 2025: Human Mobility Prediction Challenge [1]. The proposed solution models the city as a weighted directed graph, from which vector embeddings are generated to predict users' future movements. Results are evaluated using Manhattan distance and other common metrics.**

I. Introduction

A. Problem and scenario

The provided dataset consists of 4 cities (A–D), each partitioned into a 200×200 grid of cells, where each cell represents an area of 500×500 meters. For each city, a CSV file is provided containing the x and y positions of each user (identified by a numeric ID) at different time instants, defined as a combination of day ($d \in \{1, \ldots, 75\}$) and a 30-minute time window ($t \in \{0, \ldots, 47\}$). The goal of the proposed challenge is to predict the masked positions ($x, y = 999$) of a subset of 3000 users per city over days 61–75.

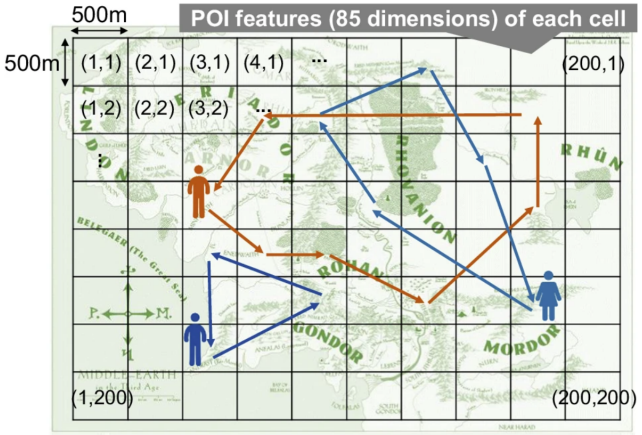

Fig. 1. City grid and user movement.

B. Why a graph-based approach

Mobility is a sequence of transitions and is therefore *naturally modeled* as a movement graph. A graph makes it possible to account for the **transition frequency** between cells (edge weight), the **directionality** of movement, and to identify movement "corridors" and spatial communities. The graph captures the empirical distribution of transitions and enables the application of *graph embedding* techniques.

C. Contributions

The work carried out consists, first of all, in the scalable construction of a *Weighted Directed Cell-Transition Graph*, followed by training cell embeddings with **Node2Vec (PecanPy)**. The inference pipeline then relies on prototypes obtained from the embeddings and on a pool of additional candidates, whose consistency is evaluated. Finally, evaluation is performed using the metrics provided by the challenge (**Manhattan** only, as **GEO-BLEU** proved to be too computationally heavy given the available hardware), as well as additional common metrics.

II. Related Work / Background

A. Graph embeddings and Node2Vec

Among graph representation learning methods, **Node2Vec** is one of the most widely used because it is *simple* and *flexible* [2]. The core idea is to generate node sequences via random walks on the graph and learn embeddings by maximizing the co-occurrence of nearby nodes in the sequence using a skip-gram model similar to *Word2Vec*. A distinctive feature of Node2Vec is the use of *second-order biased walks* (meaning that the next node selection depends on both the current node and the previous node), controlled by parameters p and q: the former regulates the tendency to return to the previous node, while the latter balances local and global exploration [2]. In practice, varying p and q yields embeddings that are more oriented toward capturing either local communities or structural roles in the graph.

B. PecanPy

Applying Node2Vec to real-scale graphs requires an efficient implementation. **PecanPy** is an optimized implementation of Node2Vec that reduces overhead and memory consumption, supporting both *weighted* and *directed* graphs and providing multiple operating modes [3]. In particular, the *SparseOTF* mode is designed for sparse graphs and performs computations on-the-fly (using less memory and being suitable for large graphs). In our case, with a cell graph of up to 40,000 nodes and a large number of observed transitions, this mode is a natural choice to keep embedding training *scalable*.

III. Dataset and Preprocessing

A. Data format and temporal split

Each record is associated with a user identifier uid, a day index d, a time slot t, and a location (x, y) on a discrete grid:

- $x, y \in \{1, \ldots, 200\}$;
- $t \in \{0, \ldots, 47\}$ (48 slots of 30 minutes);
- $d \in \{1, \ldots, 75\}$.

For each city, locations in days 61–75 are masked by setting $x = y = 999$ for a subset of 3000 users, and must be predicted [1].

During preprocessing, each city CSV is split into a training set (days 1–60) and a test set (days 61–75). In order to obtain a complete and uniformly sampled trajectory, missing (d, t) records (if any) are inferred using a forward-fill strategy, so by copying the last available location for the same user. This enforces one record per (uid, d, t) and simplifies the extraction of consecutive transitions.

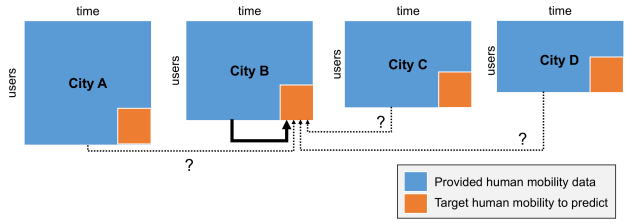

Fig. 2. Dataset composition.

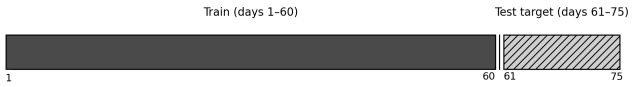

Fig. 3. Visualization of dataset split.

B. Spatio-temporal mapping

Each grid cell is mapped to a unique identifier:

$$\text{cell\_id}(x, y) = (x - 1) \cdot 200 + (y - 1), \tag{1}$$

which yields cell_id $\in [0, 39999]$. This encoding is used both for graph construction and for indexing cell embeddings.

In addition to the absolute time slot, mobility often exhibits weekly periodicity. We therefore define:

$$\text{weekday}(d) = (d - 1) \bmod 7, \tag{2}$$

$$\text{ctx}(d, t) = \text{weekday}(d) \cdot 48 + t, \tag{3}$$

obtaining ctx $\in [0, 335]$. This reduces temporal dimensionality and enables the model to capture weekly patterns.

C. Ordering and masking

Records are sorted by (uid, d, t) to preserve temporal consistency within each trajectory. Masked entries ($x = y = 999$) are tracked explicitly and excluded from any computation that requires ground truth (e.g., evaluation metrics), while remaining available as inputs for prediction.

IV. Graph Construction

This section is the core of the approach: we define and build a graph that summarizes mobility as transition frequencies.

A. Graph Definition

We construct a weighted directed graph $G = (V, E)$:

- **Nodes** V: grid cells, one node per cell_id observed in the training trajectories.
- **Directed edges** E: for each user, consecutive observations (after sorting by time) create an edge from the previous cell to the next cell.
- **Weights**: edge weight w_{uv} equals the number of times the transition $u \to v$ appears in the training data.

Self-loops ($u = v$) are removed to focus on movement rather than stationary behavior.

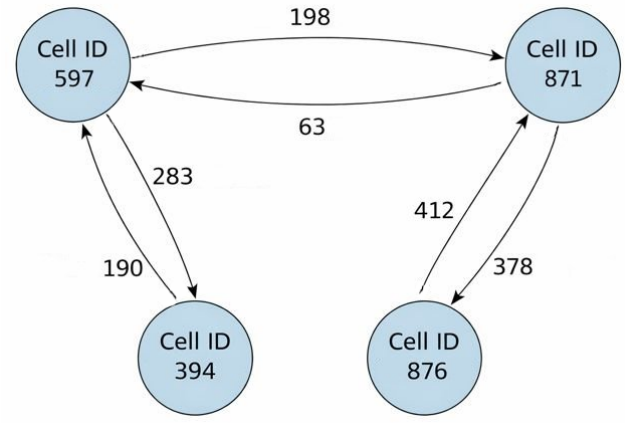

Fig. 4. Graph example: a user trajectory induces directed edges between visited cells; aggregating over users yields weighted transitions.

B. Scalable Implementation

Directly counting pairs in Python loops is slow at scale. We implement an efficient vectorized strategy:

1) Sort records by (uid, d, t).
2) Convert all (x, y) to cell_id in NumPy arrays.
3) Identify indices where consecutive rows belong to the same user.
4) Extract source and destination arrays (src, dst).
5) Remove self-loops where $src = dst$.
6) Compress pairs into a unique integer key:

$$key = src \cdot M + dst, \quad M = 200 \cdot 200, \tag{4}$$

and count occurrences.

Finally, we write a weighted edgelist file with triplets (u, v, w), which is directly consumable by embedding methods.

C. Graph Intuition and Expected Properties

The resulting graph is typically sparse and exhibits an unbalanced edge-weight distribution:

- Few transitions are extremely frequent (corridors, hubs),
- Many transitions are rare (noise or occasional trips),

- Directionality reveals asymmetric flows (e.g., home-to-work vs work-to-home).

These properties motivate weighted random walks and embedding learning: frequent edges should influence representation more strongly, but rare edges can still contribute connectivity.

V. Method

A. Overview

The pipeline has four main steps:

1) Build a weighted directed cell-transition graph from the training trajectories.
2) Learn cell embeddings with Node2Vec using PecanPy.
3) Build an embedding prototype for each (uid, ctx) pair.
4) Predict test locations by picking the best candidate cell using cosine similarity in embedding space, with fallbacks when needed.

In short: the graph represents where people usually move, embeddings turn each cell into a vector we can compare, prototypes summarize a user's typical behavior in a given time context, and inference selects a plausible cell that best matches the prototype.

B. Node2Vec Embeddings

Node2Vec learns node vectors by:

- generating biased random walks on the graph,
- treating each walk as a sequence (like a "sentence"),
- training a skip-gram model so nodes that often appear close to each other in walks get similar vectors.

At the end, we get one embedding vector per visited cell. In our case, these vectors capture how a cell sits in the mobility graph.

The walk bias is controlled by two parameters:

- p (return parameter): reduces the chance of immediately going back to the previous node,
- q (in-out parameter): controls whether walks stay local around the current node or explore further away (often explained as BFS-like vs DFS-like behavior).

Because the graph is weighted, frequent transitions are sampled more often during the walks. This means the embedding space reflects movement intensity: cells linked by strong, repeated transitions end up closer than cells connected only by rare edges.

C. PecanPy Implementation Details

We use PecanPy, which is an optimized Node2Vec implementation meant for large graphs. It offers different execution modes:

- **PreComp**: precomputes transition probabilities (more memory, faster sampling),
- **SparseOTF**: computes probabilities on-the-fly and is optimized for sparse graphs,
- **DenseOTF**: on-the-fly mode for dense graphs.

Since the cell-transition graph is sparse, we use **SparseOTF**. We train embeddings directly from the weighted edgelist. The embedding dimension, walk length, walks per node, and skip-gram window size mainly control the trade-off between modeling local neighborhoods (short-range) and capturing broader structure (long-range).

D. User-Context Prototypes

To make predictions more personalized, we build a prototype embedding for each user and time context. The goal is simple: represent, with one vector, where a user usually is on a given weekday and time slot.

In practice, for every training record we map (x, y) to a cell, take that cell's embedding, and put it into a bucket identified by (uid, ctx). Then we average all embeddings in the bucket to get the prototype vector for that user-context pair. If a user is never seen in a context (or if some cells do not have embeddings), then the prototype for that pair just does not exist. You can think of the prototype as the average point of the user's past locations in embedding space for that specific context.

E. Candidate Priors

Predicting directly over all $40{,}000$ cells is unnecessary and tends to add noise. Instead, we restrict each prediction to a small candidate set built from simple frequency priors:

- $\textbf{Top}_N(uid)$: the N most frequent cells visited by that user in training (user prior),
- $\textbf{Top}_M(ctx)$: the M most frequent cells visited overall in that context (context prior).

We take the union of the two sets. Candidates are *cell IDs*, not embeddings: embeddings are only used later to score and rank candidates.

To avoid forcing a move at every step, we also add the **previous predicted cell** ($prev$) to the candidate set, so "staying still" is always an available option. For the first test record of each user (at $d = 61, t = 0$), $prev$ is initialized as that user's last known training position (at $d = 60, t = 47$). This gives continuity between train and test and helps prevent early mistakes that would otherwise propagate over time.

F. Penalized Cosine Selection

If a prototype exists, we choose the best candidate by scoring each candidate cell embedding against the prototype embedding. The base score is cosine similarity:

$$\cos(c, p) = \frac{E_c^\top p}{\|E_c\|\|p\|}.$$

In the improved version, we add a distance-based penalty that depends on the current $prev$. For each candidate cell c, we compute the Manhattan distance from $prev$ in grid coordinates:

$$d(c, prev) = |x_c - x_{prev}| + |y_c - y_{prev}|.$$

Then we subtract a weighted penalty from the cosine score:

$$s(c) = \cos(c, p) - \lambda \cdot d(c, prev),$$

where λ controls how strongly we discourage large jumps. This acts like a soft prior that favors candidates closer to

the previous position, unless the embedding similarity clearly supports a move.

If the prototype is missing, or if none of the candidate cells has a valid embedding, we use fallbacks to always return a valid output:

1) the user's mode cell (their most frequent training cell),
2) the context mode cell (the most frequent cell in that context),
3) any node with an embedding (last resort).

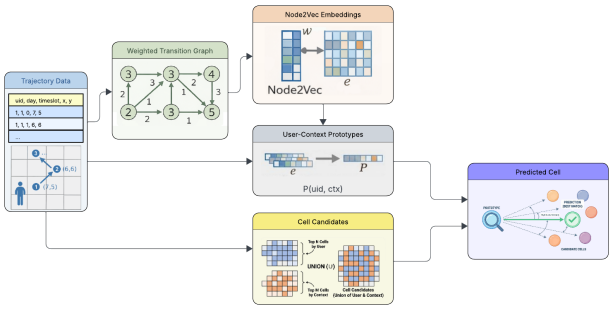

Fig. 5. Pipeline diagram.

VI. Evaluation

A. Metrics

The main metric is Manhattan distance (in grid cells):

$$L_1 = |x - \hat{x}| + |y - \hat{y}|. \tag{5}$$

We also report a few extra metrics to understand *how* the model behaves:

- **Exact match / within-k** using L_2 distance (in cells),
- L_2 **mean/median** in cells and in meters (1 cell = 500 m),
- **Move rate**: how often the predicted cell changes, compared to the real one,
- **Unique cells ratio**: how many different cells are predicted, compared to ground truth.

These help spot common failure modes, like predicting almost always the same place, jumping around too much, or collapsing to popular areas.

B. Experimental Configuration

For the evaluation run we used:

- Node2Vec: $dim = 64$, $walk\_length = 20$, $num\_walks = 10$, $window = 10$,
- PecanPy: $mode = SparseOTF$, $p = 1.0$, $q = 1.0$,
- Candidate pools: $\mathbf{Top}_{user} = 200$ and $\mathbf{Top}_{ctx} = 200$.
- Distance panalty: $\lambda = 0.05$

C. Results and Discussion

Table I summarizes the evaluation. The exact match is about 0.26, so roughly one out of four points is predicted in the correct cell. The median Manhattan error is much lower than the mean, which means errors are not "uniform": most predictions land close to the true cell, but a smaller set of cases is very wrong and pushes the average up.

TABLE I
Evaluation results.

Metric	Value
Manhattan L_1 mean (cells)	15.4507
L_1 median (cells)	4.0000
L_2 mean (cells)	11.8095
L_2 median (cells)	3.1622
L_2 mean (meters)	5904.78
L_2 median (meters)	1581.13
Exact match (within 0 cells L_2)	0.2606
Within 1 cell (L_2)	0.3520
Within 2 cells (L_2)	0.4209
Within 5 cells (L_2)	0.5824
Move rate (pred)	0.1888
Move rate (gt)	0.1747
Unique cells (pred)	10080
Unique cells (gt)	11512
Unique ratio pred/gt	0.8756

A clear issue is that the predicted move rate is higher than the real one: the method changes cell too often. This comes from the current scoring rule. The choice is driven mainly by embedding similarity plus frequency-based priors (global/context/user popularity). Even when the previous cell is included as a candidate and a small distance penalty is used, there is no strong preference to stay in the same place. So when multiple candidates get similar scores, the method can easily switch to a nearby popular cell instead of remaining stationary.

Preprocessing can also affect these results. Missing time bins are filled by carrying forward the last known position, which increases repeated observations of the same cell. This tends to strengthen popularity signals and makes some cells appear very frequent in the training data. However, if self-loops are removed when building the transition graph, these repeated "stay" periods do not become explicit stay-transitions in the graph, so the effect is mostly indirect (through prototypes and popularity counts). Also, long real gaps are compressed into single jumps after filling, which hides intermediate movement and can make some transitions look more abrupt and harder to predict. This is a plausible source of the large-error outliers.

Overall, the method works well when the true next cell is among the frequent candidates for that user/context, which explains the good median error. The heavy tail (mean much larger than the median) likely happens when the true next cell is missing from the candidate set, or when embedding similarity is not a good indicator of the next step for that specific situation.

D. Future Work

The current baseline already provides reasonable predictions, but the results show two clear limitations: the model sometimes produces large jumps, and it changes cell too often compared to the ground truth. Future improvements will

mainly focus on adding stronger short-term continuity, without changing the overall graph-based approach.

A first direction is to include a transition-based prior from the previous cell. Instead of generating candidates only from user and context popularity, we can also add the top-K most frequent outgoing neighbors of the last observed (or last predicted) cell in the training graph. This keeps the candidate set small, but makes it more consistent with how users actually move on the grid.

Second, we plan to introduce a simple inertia mechanism to reduce unnecessary switching. For example, the previous cell can always be included as a candidate and receive a small bonus, so that the model stays still when multiple options have very similar similarity scores. This should reduce the predicted move rate and improve stability in dense areas where many nearby cells look similar in embedding space.

Finally, we can tune the Node2Vec hyperparameters (walk length, number of walks, window size, and the p/q bias parameters) to better match the mobility graph structure. Since embeddings drive most of the decision rule, even small improvements in representation quality can translate into better ranking of candidates at inference time.

VII. Conclusion

This work presents a graph-based baseline for grid mobility forecasting. The core idea is building a weighted directed cell-transition graph and using it for both embedding learning and inference. Node2Vec embeddings trained with PecanPy capture similarities between cells based on how people move. For prediction, we combine user-context prototypes with simple popularity priors, then pick the best candidate using cosine similarity in embedding space, penalizing cells that are too distant.

References

[1] T. Yabe, K. Tsubouchi, and T. Shimizu, "Sigspatial giscup 2025 human mobility prediction challenge: Open data," Dec. 2025. [Online]. Available: https://zenodo.org/records/18050604

[2] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, 2016, pp. 855–864. [Online]. Available: https://doi.org/10.1145/2939672.2939754

[3] R. Liu and A. Krishnan, "Pecanpy: a fast, efficient and parallelized python implementation of node2vec," *Bioinformatics*, vol. 37, no. 19, pp. 3377–3379, 2021. [Online]. Available: https://doi.org/10.1093/bioinformatics/btab202