Dynamic FastMap: An Efficient Algorithm for Spatiotemporal Embedding of Dynamic Graphs*

Omkar Thakoor
Department of Computer Science
University of Southern California
othakoor@usc.edu

T. K. Satish Kumar
Information Sciences Institute
University of Southern California
tkskwork@gmail.com

Abstract

Efficiently embedding graphs in a Euclidean space has many benefits: It allows us to interpret and solve graph-theoretic problems using geometric and analytical methods. It also allows us to visualize graphs and support human-in-the-loop decision-making systems. FastMap is a near-linear-time graph embedding algorithm that has already found many real-world applications. In this paper, we generalize FastMap to Dynamic FastMap, which efficiently embeds dynamic graphs, i.e., graphs with time-dependent edge-weights, in a spatiotemporal space with a user-specified number of dimensions, while reserving one dimension for representing time. Through a range of experiments, we also demonstrate the efficacy of Dynamic FastMap as an algorithm for spatiotemporal embedding of dynamic graphs.

Introduction

In many real-world problem domains, the underlying environment can be modeled as a graph. Moreover, in many such domains, these graphs are dynamic wherein the edge-weights evolve with time. A vital aid in decision making in dynamic environments is the ability to anticipate how the graph changes by continually visualizing and monitoring it. Towards that end, we require an efficient algorithm for spatiotemporal embedding of dynamic graphs.

In this paper, we build on a near-linear-time graph embedding algorithm called FastMap (Cohen et al. 2018; Li et al. 2019). FastMap embeds the vertices of a given undirected graph in a Euclidean space while approximately preserving the shortest-path distances as Euclidean distances for all pairs of vertices. The algorithm is itself based on a Data Mining algorithm, also called FastMap (Faloutsos and Lin 1995), that is designed for efficiently embedding complex and/or abstract objects, such as long DNA strings, multi-media datasets like voice excerpts or images, or medical datasets like MRIs, in a Euclidean space by leveraging a domain-specific distance function between pairs of objects. Despite the efficiency of FastMap for embedding undirected graphs, by design, it is only applicable to static graphs.

In this paper, we present a dynamic version of FastMap, called Dynamic FastMap, for embedding dynamic graphs: One dimension represents time, while the other dimensions represent pairwise distances between vertices of the graph at any timestep. The naive approach of invoking FastMap independently at each timestep does not satisfactorily create a coherent spatiotemporal embedding. This is because, even with small changes to the edge-weights across consecutive timesteps, FastMap can produce large differences in the coordinates created for a vertex. Although Dynamic FastMap also invokes FastMap at every timestep, it alleviates the foregoing issue by solving a "patching" problem between the FastMap embeddings produced at consecutive timesteps. The patching problem is to find an optimal linear transformation of the FastMap embedding produced at timestep t that minimizes the "difference" relative to the embedding at timestep $t-1$. It can be efficiently solved using Gurobi, a state-of-the-art optimizer developed in the Operations Research community. Through a range of experiments, we demonstrate the efficacy of Dynamic FastMap.

FastMap

FastMap (Faloutsos and Lin 1995) embeds a collection of abstract objects in an artificially created Euclidean space to enable geometric interpretations, algebraic manipulations, and downstream Machine Learning algorithms. It gets as input a collection of abstract objects $\mathcal{O}$, where $D(O_i, O_j)$ represents the domain-specific distance between objects $O_i, O_j \in \mathcal{O}$. A Euclidean embedding assigns a κ-dimensional point $p_i \in \mathbb{R}^\kappa$ to each object O_i. A good Euclidean embedding is one in which the Euclidean distance χ_{ij} between any two points p_i and p_j closely approximates $D(O_i, O_j)$. For $p_i = ([p_i]_1, [p_i]_2 \dots [p_i]_\kappa)$ and $p_j = ([p_j]_1, [p_j]_2 \dots [p_j]_\kappa)$, $\chi_{ij} = \sqrt{\sum_{r=1}^{\kappa} ([p_j]_r - [p_i]_r)^2}$.

FastMap creates a κ-dimensional Euclidean embedding for a user-specified value of κ. In the very first iteration, FastMap heuristically identifies the farthest pair of objects O_a and O_b in linear time. Once O_a and O_b are determined, every other object O_i defines a triangle with sides of lengths $d_{ai} = D(O_a, O_i)$, $d_{ab} = D(O_a, O_b)$ and $d_{ib} = D(O_i, O_b)$, as shown in Figure 1 (top panel). The sides of the triangle define its entire geometry, and the projection of O_i onto the line $\overline{O_a O_b}$ is given by

$$x_i = (d_{ai}^2 + d_{ab}^2 - d_{ib}^2)/(2d_{ab}). \quad (1)$$

FastMap sets the first coordinate of p_i, the embedding of O_i, to x_i. In the subsequent $\kappa - 1$ iterations, the same pro-

*DARPA grant HR001120C0157 and NSF grant 2112533

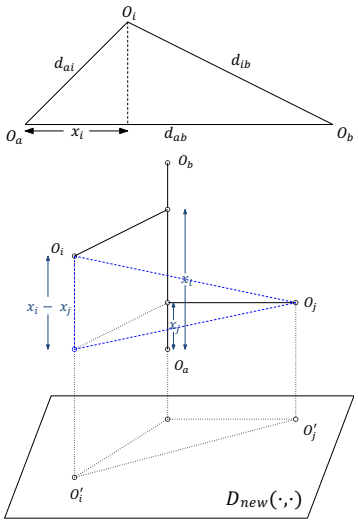

Figure 1: Illustrates how coordinates are computed and recursion is carried out in FastMap, borrowed from (Cohen et al. 2018). The top panel illustrates the "cosine law" projection in a triangle. The bottom panel illustrates the process of projecting onto a hyperplane that is perpendicular to $\overline{O_aO_b}$.

cedure is followed for computing the remaining $\kappa - 1$ coordinates of each object. However, the distance function is adapted for different iterations. For example, for the first iteration, the coordinates of O_a and O_b are 0 and d_{ab}, respectively. Because these coordinates fully explain the true domain-specific distance between these two objects, from the second iteration onward, the rest of p_a and p_b's coordinates should be identical. Intuitively, this means that the second iteration should mimic the first one on a hyperplane that is perpendicular to the line $\overline{O_aO_b}$, as shown in Figure 1 (bottom panel). Although the hyperplane is never constructed explicitly, its conceptualization implies that the distance function for the second iteration should be changed for all i and j in the following way:

$$D_{new}(O'_i, O'_j)^2 = D(O_i, O_j)^2 - (x_i - x_j)^2. \qquad (2)$$

Here, O'_i, O'_j are the projections of O_i, O_j, respectively, onto this hyperplane, and $D_{new}(\cdot,\cdot)$ is the new distance function.

FastMap can also be used to embed the vertices of a given graph $G = (V, E)$ in a Euclidean space so as to preserve the pairwise shortest-path distances between them. As such, the Data Mining FastMap algorithm cannot be directly used for generating a graph embedding in linear time, since it assumes that the distance d_{ij} between two objects O_i and O_j can be computed in "constant time", i.e., independent of the number of objects in the problem domain, whereas, computing the shortest-path distance between two vertices depends on the size of the graph. This challenge is tackled in (Li et al. 2019) to build a graph-based version of FastMap that runs in near-linear time. There, the key idea is to root shortest-path trees at vertices O_a and O_b, in each iteration, to yield all necessary distances d_{ai} and d_{ib} in one shot, achieving near-constant amortized time complexity.

FastMap is presented in lines 4-23 of Algorithm 1.

Algorithm 1: Dynamic FastMap

Input: $\{G^0, G^1, \ldots, G^T\}$, κ, and ϵ.
Output: $Z^t = [z_1^t, z_2^t, \ldots, z_N^t]$, where $z_i^t \in \mathbb{R}^\kappa$ for all $i = 1, 2, \ldots, N$ and $t = 0, 1, \ldots, T$.

1 $Z^0 = \textsc{FastMap}(G^0)$;
2 **for** $t = 0, 1, \ldots, T$ **do**
3 $Z^t = \textsc{Patch}(Z^{t-1}, \textsc{FastMap}(G^t, \kappa, \epsilon))$
4 **Function** `FastMap` $(G = (V, E), \kappa, \epsilon)$ **:**
5 **for** $r = 1, 2, \ldots, \kappa$ **do**
6 Choose $v_a \in V$ randomly and let $v_b = v_a$;
7 **for** $t = 1, 2 \ldots 10$ **do**
8 $\{d_{ai} : v_i \in V\} \leftarrow$ $\textsc{ShortestPathTree}(G, v_a)$;
9 $v_c \leftarrow \text{argmax}_{v_i}\{d_{ai}^2 - \sum_{j=1}^{r-1}([p_a]_j - [p_i]_j)^2\}$;
10 **if** $v_c == v_b$ **then**
11 Break;
12 **else**
13 $v_b \leftarrow v_a$; $v_a \leftarrow v_c$;
14 $\{d_{ai} : v_i \in V\} \leftarrow \textsc{ShortestPathTree}(G, v_a)$;
15 $\{d_{ib} : v_i \in V\} \leftarrow \textsc{ShortestPathTree}(G, v_b)$;
16 $d'_{ab} \leftarrow d_{ab}^2 - \sum_{j=1}^{r-1}([p_a]_j - [p_b]_j)^2$;
17 **if** $d'_{ab} < \epsilon$ **then**
18 Break;
19 **for** *each* $v_i \in V$ **do**
20 $d'_{ai} \leftarrow d_{ai}^2 - \sum_{j=1}^{r-1}([p_a]_j - [p_i]_j)^2$;
21 $d'_{ib} \leftarrow d_{ib}^2 - \sum_{j=1}^{r-1}([p_i]_j - [p_b]_j)^2$;
22 $[p_i]_r \leftarrow (d'_{ai} + d'_{ab} - d'_{ib})/(2\sqrt{d'_{ab}})$;
23 **return** $[p_1, p_2, \ldots, p_N]$;

The Dynamic FastMap Algorithm

We present Dynamic FastMap in Algorithm 1. It takes as input snapshots of a dynamic graph on N vertices. Each snapshot G^t, for $t = 0, 1, \ldots, T$, describes the graph at timestep t. Additionally, the algorithm takes as input κ, the number of spatial dimensions for the embedding, and ϵ, a threshold parameter required for invoking FastMap as a subroutine. Algorithm 1 outputs the sequence of embeddings $Z^t = [z_1^t, z_2^t, \ldots, z_N^t]$, where each $z_i^t \in \mathbb{R}^\kappa$ denotes the κ-dimensional point assigned to vertex i at timestep t, for $i = 1, 2, \ldots, N$ and $t = 0, 1, \ldots, T$.

Algorithm 1 iterates through the timesteps in lines 2-3. In the first iteration corresponding to timestep $t = 0$, it obtains the FastMap embedding Z^0 of G^0. In each subsequent iteration corresponding to timestep $t > 0$, it first invokes FastMap on G^t to obtain an embedding $X^t = [x_1^t, x_2^t, \ldots, x_N^t]$, where each $x_i^t \in \mathbb{R}^\kappa$ denotes the κ-dimensional point assigned to vertex i. The algorithm then modifies X^t to obtain Z^t. The modification is done by patching X^t relative to the previous embedding Z^{t-1}.

The process of patching intends to obtain a smooth transition of the points between the embeddings Z^{t-1} and Z^t. Specifically, the idea is to obtain Z^t from X^t while preserving all the pairwise distances between its points and minimizing the difference between Z^t and Z^{t-1}. Any operation

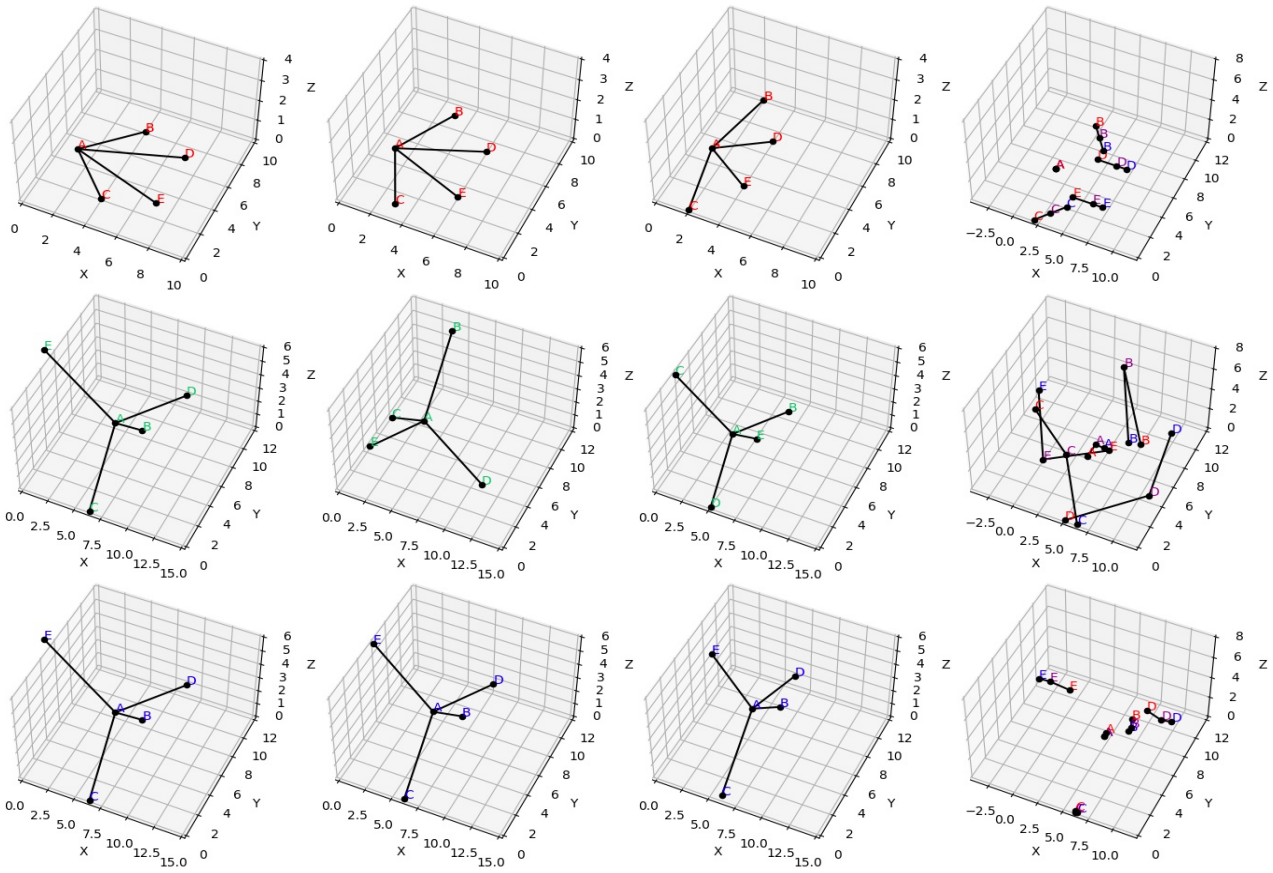

Figure 2: Illustrates the working of Dynamic FastMap. The first three panels in the top row show the ground truth, i.e., a dynamic graph on 5 vertices, colored red, at $t = 0$, 1, and 2, respectively. The fourth panel shows the trajectories of individual vertices, with red, purple, and blue representing $t = 0$, 1, and 2, respectively. The panels in the second and third rows are in correspondence with the first row, but show the FastMap embedding X^t in green and the Dynamic FastMap embedding Z^t in blue, respectively. Dynamic FastMap produces trajectories that are significantly closer to the ground truth.

that preserves pairwise distances is referred to as an isometric operation. Examples include rotation, reflection, and translation. Here, we broadly consider our isometric operation to be a combination of a (linear) orthogonal transformation and a (linear) translation. An orthogonal transformation generalizes both rotation and reflection. Moreover, the difference between Z^t and Z^{t-1} can be quantified using the sum of N squared L_2 norms $\sum_{i=1}^{N} \|z_i^t - z_i^{t-1}\|^2$.

We let $z_i^t = A^t x_i^t + b^t$, where A^t represents an orthogonal transformation and b^t represents a translation. A^t is a $\kappa \times \kappa$ matrix of unknowns, and b^t is a κ-dimensional vector of unknowns. Patching solves the optimization problem:

$$\begin{aligned} \min_{A^t, b^t} \quad & \sum_{i=1}^{N} \|A^t x_i^t + b^t - z_i^{t-1}\|^2 && (3) \\ \text{s.t.} \quad & \|A_i^t\|^2 = 1 && \forall\, 1 \le i \le \kappa, \\ & A_i^t \cdot A_j^t = 0 && \forall\, 1 \le i < j \le \kappa. \end{aligned}$$

Here, A_i^t denotes the i^{th} column of A^t, and the constraints enforce the orthogonality of A^t.

This optimization problem has $\kappa^2 + \kappa$ unknowns, independent of N, and with κ typically equal to 2 or 3 for visualization. Therefore, it can be solved efficiently using a state-of-the-art optimizer such as Gurobi. Figure 2 shows that solving the patching problem imparts smooth trajectories to the points between embeddings produced by Dynamic FastMap.

Experimental Results

In this section, we present experimental results. We implemented all algorithms and experimentation procedures using Python3 with the NetworkX library. For the FastMap subroutine in Algorithm 1, we used $\kappa = 3$ and $\epsilon = 10^{-4}$. All experiments were conducted on a laptop with a 1.6GHz Intel Core i5 processor and 8GB 1600MHz DDR3 memory.

We generated dynamic graphs derived from static graphs obtained from the benchmark datasets DIMACS, Small World, and the Waxman graph generator. Each benchmark instance serves as the initial graph G^0. For $0 < t \le T$, G^t is derived from G^{t-1} by perturbing each edge-weight to be within $[1 - \delta, 1 + \delta]$ times its previous value. The DIMACS graphs were obtained from http://networkrepository.com/dimacs.php and https://mat.tepper.cmu.edu/COLOR/instances.html, and the Small World graphs were generated

Instance	($\delta = 0.02$)		($\delta = 0.05$)		($\delta = 0.1$)		($\delta = 0.2$)	
	FM	DFM	FM	DFM	FM	DFM	FM	DFM
queen8_8	2891.2	697.3	2956.9	572.6	2515.1	653.3	2160.4	1025.7
david	53.7	4.1	5249.7	2814.4	5648.7	1604.4	9346.4	131.6
miles1000	12828.3	388.7	3951.1	797.8	1375.4	103.0	8641.3	371.5
anna	36655.7	11.5	5068.9	1548.2	32443.5	836.1	17549.3	4583.8
queen14_14	1798.3	1529.0	2973.8	988.1	2095.0	1255.2	2222.8	920.7
n0100k4p0.3	2694.8	973.3	1990.7	958.8	2192.0	821.1	3406.8	811.9
n0100k4p0.6	2125.5	821.9	1898.3	840.5	2635.0	396.5	1472.9	622.7
n0100k6p0.3	1253.6	534.6	1596.9	323.4	1663.5	405.5	1082.1	544.4
n0100k6p0.6	1727.0	418.3	613.3	274.1	1012.8	366.2	610.1	364.0
n0200k6p0.6	2358.7	1554.0	2494.3	1796.1	2231.2	1155.8	1500.5	999.6
wx(10,0,8,0.8)	741.2	2.0	309.4	6.4	830.4	24.6	1258.5	80.6
wx(25,0.4,0.8)	1692.0	291.3	3255.5	185.4	3372.4	316.1	2017.6	599.3
wx(25,0.8,0.4)	4237.5	445.6	4105.2	372.2	5099.8	750.9	2880.6	584.3
wx(25,0.4,0.4)	3613.2	1071.7	3758.8	1557.6	2971.3	1435.0	3629.3	766.3
wx(100,0.4,0.8)	527.1	151.3	124.6	115.9	406.6	137.2	730.2	162.8

Instance	(T = 1)		(T = 2)		(T = 4)		(T = 8)	
	FM	DFM	FM	DFM	FM	DFM	FM	DFM
queen8_8	321.0	261.9	1133.7	439.8	2224.1	710.7	3896.6	1248.9
david	1267.1	229.5	3273.5	775.7	5858.1	2189.4	15465.1	4575.9
miles1000	635.4	231.2	1231.8	415.0	1726.9	621.4	5676.5	1185.7
anna	4002.6	348.9	6881.8	830.8	9827.0	2239.7	25034.9	4649.9
queen14_14	800.1	464.9	1593.7	796.5	3093.4	1382.7	5426.3	2971.6
n0100k4p0.3	750.6	180.3	1568.2	369.9	2876.6	717.7	4835.5	1183.8
n0100k4p0.6	519.8	157.1	843.7	232.7	1611.3	502.1	3110.5	1030.7
n0100k6p0.3	169.0	160.0	511.7	290.3	1126.5	546.9	2530.3	1041.7
n0100k6p0.6	288.6	136.2	391.0	233.2	861.8	346.3	1884.4	767.6
n0200k6p0.6	643.8	354.7	1053.5	742.0	1860.8	1345.1	3843.1	2524.7
wx(10,0.8,0.8)	7.9	0.6	13.0	1.3	44.9	4.6	1331.6	107.4
wx(25,0.4,0.8)	152.2	139.7	285.0	264.8	2210.3	279.3	4287.7	645.5
wx(25,0.8,0.4)	458.2	193.9	621.9	324.4	883.0	506.2	3617.7	766.0
wx(25,0.4,0.4)	2102.3	205.1	4302.8	568.0	7745.4	1236.5	11456.5	2450.9
wx(100,0.4,0.8)	176.7	26.8	349.9	56.8	657.8	105.9	1208.3	234.9

Table 1: Shows the superior performance of Dynamic FastMap (DFM) over FastMap (FM). Entries in both tables indicate the objective value in Equation (3) on various benchmark instances. δ is a perturbation parameter used for changing the edge-weights across timesteps. In the left table, we vary δ, fixing $T = 5$. In the right table, we vary T, fixing $\delta = 0.1$.

using the Newman-Watts-Strogatz method in NetworkX. In both cases, each edge-weight was chosen uniformly at random from the interval $[1, 10]$. The names of the Small World graphs indicate the parameter values of N, the number of vertices, k, the number of neighbors in a ring, and p, the probability of adding a new edge. The Waxman generator is widely used for communication networks. A Waxman graph 'wx(N, α, β)' has N nodes, where the parameter α controls the ratio of short edges to long edges and the parameter β controls the overall density of the graph.

Table 1 shows the comparative performances of Dynamic FastMap and FastMap on 5 representative instances from each of the benchmark suites. The left table varies δ while fixing T and clearly shows that Dynamic FastMap significantly outperforms FastMap on a measure of stability quantified by Equation (3). The right table varies T while fixing δ and shows similar results.

Conclusions

In this paper, we considered the problem of embedding dynamic graphs in a spatiotemporal space to support visualization and human-in-the-loop decision making. We presented an efficient algorithm, called Dynamic FastMap, that builds on the efficient FastMap algorithm for embedding static graphs. Dynamic FastMap invokes FastMap at every timestep but also solves a critical patching problem between consecutive timesteps. We showed that the patching problem can also be solved efficiently, imparting viability to Dynamic FastMap. Through experiments, we demonstrated the efficacy of Dynamic FastMap compared to FastMap. In future work, we will apply Dynamic FastMap to understand and predict the behavior of dynamic graphs in communication networks, social networks, and traffic networks.

References

Cohen, L.; Uras, T.; Jahangiri, S.; Arunasalam, A.; Koenig, S.; and Kumar, T. K. S. 2018. The FastMap algorithm for shortest path computations. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*.

Faloutsos, C., and Lin, K.-I. 1995. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*.

Li, J.; Felner, A.; Koenig, S.; and Kumar, T. K. S. 2019. Using FastMap to solve graph problems in a Euclidean space. In *Proceedings of the International Conference on Automated Planning and Scheduling*.