ABOUT GRAPH DEGENERACY, REPRESENTATION LEARNING AND SCALABILITY

Simon Brandeis
CentraleSupelec
France
simon.brandeis@student.ecp.fr

Adrian Jarret
CentraleSupelec
France
adrian.jarret@student.ecp.fr

Pierre Sevestre
CentraleSupelec
France
pierre.sevestre@student.ecp.fr

May 19, 2020

ABSTRACT

Graphs or networks are a very convenient way to represent data with lots of interaction. Recently, Machine Learning on Graph data has gained a lot of traction. In particular, vertex classification and missing edge detection have very interesting applications, ranging from drug discovery to recommender systems. To achieve such tasks, tremendous work has been accomplished to learn embedding of nodes and edges into finite-dimension vector spaces. This task is called *Graph Representation Learning*. However, Graph Representation Learning techniques often display prohibitive time and memory complexities, preventing their use in real-time with business size graphs. In this paper, we address this issue by leveraging a degeneracy property of Graphs - the *K-Core Decomposition*. We present two techniques taking advantage of this decomposition to reduce the time and memory consumption of walk-based Graph Representation Learning algorithms. We evaluate the performances, expressed in terms of quality of embedding and computational resources, of the proposed techniques on several academic datasets. Our code is available at https://github.com/SBrandeis/k-core-embedding.

***K*eywords** Graph · Network · Embeddings · Degeneracy · Learning

1 Introduction

1.1 Context and motivations

Graph embeddings, *i.e.* graph representations into vector spaces, are a useful representation for many downstream machine learning applications. However, it is crucial to determine what makes a "good" embedding. There exists two main qualities that are often considered as a target for a good embedding when it comes to graphs : structural equivalence and k^{th}-order similarity. Structural equivalence states that two vertices of the graph that see the same (or a similar) neighbourhood should have similar embeddings, with respect to the embedding of their common neighbourhood. k^{th}-order similarity is the notion of k^{th}-neighbour. It simply considers that the embeddings should conserve the proximity in the vector space that existed in the graph. Two nodes related with a short path should be quite close in the embeddings space.

Such things been stated, we also should emphasize that finding such good quality embeddings is a time consuming task. The main focus of our paper is to study ways to speedup the execution time of the embedding generation procedure, while maintaining a good embeddings quality.

1.2 Mathematical framework

Let us introduce the mathematical notations we will use.

1.2.1 General notations

Let's consider a Graph (*a.k.a.* Network) G, composed of a set of vertices or nodes V and a set of edges $E \subset V^2$:

$$G = (V, E) \tag{1}$$

We will denote by v an element of V, that is a vertex or node of G. The symbol e will denote an element of E, that is an edge of G. Each element e of E represents a connection between two nodes, and as such can be represented as a pair (u, v) of elements of V. This connection can be unidirectional, meaning the order in the pair (u, v) matters. We will call such connections *directed* edges and will note them: $(u \to v)$. *A contrario*, an edge can be bidirectional and we will call it *undirected*. A graph will be said *directed* or *undirected* if all of its edges are respectively directed or undirected. Moreover, edges in a Graph can have a relative importance. We will model this by assigning to each edge (u, v) a numerical *weight* $w_{u,v}$. A graph presenting such kind of edges will be called a *weighted* graph. When all edges are equivalent, the graph is said to be *unweighted.*

We can define an inclusion relationship between two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ as follows:

$$\begin{aligned} G_1 \subset G_2 \iff & V_1 \subset V_2 \\ & \text{and} \quad E_1 \subset E_2 \\ & \text{and} \quad E_1 \subset V_1 \times V_1 \end{aligned} \tag{2}$$

G_1 is then said to be a *subgraph* of G_2. The other way around, G_2 is a *supergraph* of G_1.

To quantify the connectivity of a node or vertex, we use the notion of *degree* of a node. In the case of undirected and unweighted graphs, the degree of a node is simply the number of edges including this node:

$$\forall v \in V \qquad \deg(v) = |\{(u, v) \quad \forall u \in V : (u, v) \in E\}| \tag{3}$$

Where $|.|$ denotes the cardinality of the set.

The degree of a node can be generalized to directed graph by counting in-bound and out-bound edges separately. The number of in-bound edges and out-bound edges are then respectively called *in-degree* and *out-degree* of the node. When the edges are weighted, the degree is defined to be the sum of the weights corresponding to the aforementioned edges. We can represent the degree of each node in the graph via the diagonal *degree matrix* $D = (d_{u,v})_{(u,v)\in V\times V}$ of size $|V| \times |V|$:

$$d_{u,v} = \begin{cases} \deg(u) & \text{if} \quad u = v \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

A graph G can be represented by its adjacency matrix A, of size $|V| \times |V|$, that defines the connections between the nodes of G. An edge of weight $w_{u,v}$ exists from node u to node v if and only if the element $A_{u,v}$ of coordinates u, v is equal to $w_{u,v}$:

$$\begin{aligned} \forall (u, v) \in V \times V \qquad A_{u,v} = w_{u,v} \neq 0 &\iff (u \xrightarrow{w_{u,v}} v) \in E \\ = 0 &\iff (u \to v) \notin E \end{aligned} \tag{5}$$

1.2.2 Graph Representation Learning

Graph Representation learning is the task of learning a "good" representation function f that maps each node v of the Graph to a representation in a n-dimensional vector space. That is, find:

$$f : V \to \mathbb{R}^n \tag{6}$$

that has relevant properties for the task we want to achieve. We can extend this definition to the set of edges E of the graph.

The image of V through the mapping f will be called an *embedding* of the graph. We will denote it X_V. Its elements will be noted x_v, v living in V.

$$\begin{aligned} X_V &= im_f(V) \\ &= \{x_v = f(v) \quad \forall v \in V\} \subset \mathbb{R}^n \end{aligned} \tag{7}$$

As mentioned above, there exist plenty of criterions to evaluate the quality of an embedding. Although, metrics such as the k-th order proximity are hard to verify in the embedded vector space. In order to quantify the quality of the information from the graph retained in the embedding, we trained a logistic regression to solve a downstream classification task - *e.g.* node classification or missing edge detection - on the embedded space. The quality of the embedding is then measured by the F_1 score of the classifier:

$$F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{8}$$

1.2.3 Degeneracy and k-core decomposition

Another key concept of our study is the degeneracy of graphs. It is a way to measure the density of graphs. Let k be an integer, a graph G is said to be k-degenerate if all its subgraphs have a vertex of degree at most k. The smallest k for which the graph is k-degenerate is called the *degeneracy* of the graph, and we'll note it $k_{degeneracy}$.

In addition, for every integer k lower than $k_{degeneracy}$, we can define the *k-core* of the graph as the *maximal* (in the sense of the inclusion relationship defined in equation 2) *connected sub-graph* of G where all vertices have a degree of at least k. The degeneracy of the graph is then the largest k such that the k-core is not empty.

$$\begin{aligned} &\forall k \leq k_{degeneracy}, \quad \text{core}_G(k) \subset G \\ &\forall v \in \text{core}_G(k), \quad \deg(v) \geq k \\ &\forall \text{subgraph} \subset G \quad \text{such that:} \quad \forall v \in \text{subgraph} \quad \deg(v) \geq k, \quad \text{subgraph} \subset \text{core}_G(k) \end{aligned} \tag{9}$$

A node of the graph G is said to have a *core number* or *core index* of $c \in \mathbb{N}$ when it is a node of the c-core but not of the $(c+1)$-core of G.

The k-cores of a graph are a way to decompose a graph in a hierarchical sequence of smaller and denser sub graphs. The so-called *k-core decomposition* has been used notably for massive graph visualization [1] and Graph structure analysis[2] [3] purposes.

1.2.4 Random walk in a graph

Finally, we need to introduce the notion of random walk in a graph. Given a node v of the graph, we can define a stochastic process W_v such that:

$$\begin{aligned} W_v^0 &= v \\ \forall t \in \mathbb{N} \quad W_v^t &\in \text{neighbours}(W_v^{t-1}) \end{aligned} \tag{10}$$

Where neighbours(u) is the set of neighbours of a node u, that is the set of nodes that have a direct connection to the node u:

$$\forall u \in V, \quad \text{neighbours}(u) = \{v \in V : (u \to v) \in E\} \tag{11}$$

A random walk of length l and rooted in v is a sequence of realizations of W_v:

$$(W_v^0 = v, W_v^1, ..., W_v^{l-1}) \tag{12}$$

Random walks have been used for community detection in graphs [4] or for content recommendation [5].

1.3 Related Work

There exists roughly three families of graph representation learning techniques [6]. Factorization-based methods use the adjacency matrix to embed vectors. Random-walk based methods extract context from a node's neighbourhood with random walks and use methods inspired from Natural Language Processing to compute nodes' embeddings. More recently, new methods leveraging neural networks have been proposed to tackle the task of graph embeddings. Due to the lack of a consensual benchmark dataset and evaluation process, it is hard to objectively compare the quality of the embedding produced by those methods.

1.3.1 Factorization-based methods

Factorization-based methods embed the graph by factorizing a matrix representing the graph. Matrices used are often the adjacency matrix A or the laplacian matrix $L = D - A$. Works worth citing include Locally-Linear Embeddings or LLE[7], Laplacian Eigenmaps[8], GraRep[9], HOPE[10]. All cited methods have a time complexity in $O(|E| \times n^2)$ - n being the dimension of the embedding space- except for GraRep that has a time complexity in $O(|V|^3)$.

Factorization-based methods require the factorization of often very large matrices. Indeed, real-life graphs often display tens of thousands of nodes.

1.3.2 Random walks based Frameworks

Random walks (see equations 10 to 12) based embedding methods were introduced by [11]. They are based on a simple assessment : the distribution law of vertices appearing in short random walks in the graph follows a power law, much like the distribution of words in natural language. This parallelism inspired [11] to transpose word embedding techniques such as the SkipGram model [12] to nodes in a graph.

The SkipGram model uses neural network to predict a word, given its context (*e.g.*, a sentence). The idea behind [11] is to extract the context of each node by doing random walks in the graph, thus building "sentences" of nodes. If we run a sufficient number of walks for each node in the graph, the "node corpus" is supposed to hold enough context information for every node in the graph. Then, the SkipGram model is finally applied to build embeddings of nodes using the random walks as their context, the same way it builds embeddings of words with sentences as the context. This framework is known as DeepWalk.

Numerous papers proposed tweaks to enhance the embedding quality of DeepWalk, the most notable certainly bieng Node2Vec [13]. The main contribution of Node2Vec is the introduction of two hyperparameters allowing to favour breadth-first sampling (BFS) or depth-first sampling (DFS) at each step of the random walk generation, leading to a significant increase in learning quality representations in complex networks.

Random walk based techniques display a time complexity in $O(|V| \times n)$[6].

1.3.3 Graph auto-encoders

Deep neural networks are well-known for their ability to learn useful representations of structured data, *e.g.* images. Graphs form no exception to this, and the use of deep neural networks to address network-structured data has gained a lot of traction in the last five years.

Structural deep network embedding (SDNE) [14] and Deep neural graph representation (DNGR) [15] (time complexities of respectively $O(|V| \times |E|)$ and $O(|V|^2)$) leverage deep auto-encoders to learn a low-dimensional representation of every node's neighbourhood, *i.e.* its corresponding row the adjacency matrix A.

Kipf at al.[16] introduced the notion of convolution over a graph, or Graph Convolutional Networks (GCN), later improved by adding an attention mechanism[17]. GCNs have been later used as encoders for Graph Auto-Encoders or GAE [18][19] and in the GraphSAGE algorithm [20]. Convolution-based embeddings display a time complexity in $O(|E| \times n^2)$

1.3.4 Propagation based Frameworks

The previous methods have time complexities at least linear with respect to the number of nodes or edges of the graph. While they provide satisfactory results in the task of learning embeddings, they can become impracticable when applied to real life networks having millions of vertices and edges. In consequence, people have work on frameworks to accelerate the whole process of extracting embeddings.

Given an input graph to embed, the idea introduced in the HARP framework [21] is to first extract a sequence of subgraphs of decreasing sizes. Each subgraph is a coarsened version of the previous one, obtained by merging nodes that share some common characteristics. A small number of iterations leads to really small graphs: 6 to 8 iterations reduce the size of the input graph by 90 to 95 %. The next step is to use any embedding method to create a representation of the smallest graph, which can be done quite fast and efficiently as it has a reasonable shape. Finally, the embeddings are propagated back to the successive subgraphs. The agnosticism of this framework to the underlying method makes it very interesting and flexible.

The MILE framework [22] refined the general process by taking into account weighted edges in the graph coarsening, and by propagating the obtained embeddings using a graph convolutional neural network.

These two frameworks have made the computation of graph embeddings easier for very large graphs, way larger than what was possible with direct graph embedding techniques. As an example, MILE can comfortably scale to graph with 9 million nodes and 40 million edges, where direct methods run out of time and memory on a modern workstation.

2 Proposed Methods

The main objective of our work was to take advantage of the core decomposition of graphs to speed up graph embedding techniques. We propose and experimentally evaluate two different methods that we designed. We shall notice that all the embedding strategies presented later on apply to connected graphs. As our methods are only based on the structure of the graphs, each connected subgraph of a given graph are independent. For the sake of simplicity, we will always consider the largest connected subgraph. Consequently, we will only consider graph where the 0-core is equal to the 1-core. Indeed, nodes in the 0-core without being in the 1-core are nodes that do not have any connection. It is easy to remark that we cannot extract any information from the context of these nodes (they have an empty context) and so proposing a structural-based vector representation for them makes no sense.

2.1 Core-Adaptive Random Walks

DeepWalk[11] and similar embedding techniques use random walks to explore the context of nodes in the graph and extract structural information from it. The intricacy of a vertex's context is strongly related to its degree: indeed, a node with a high degree (a high number of neighbours) have a more complex "context" than a node with only a couple neighbours. This intuition can be qualitatively verified on random walks rooted from the node: the generated random walks will be much more similar to one another when the degree of the node is lower.

We believe that the higher a node's *core index*, the more intricate its context is, and thus more random walks are necessary to extract meaningful structural information from it. Indeed, a node with a high core index is located in a dense portion of the graph: not only it has lots of connections (a high degree), but its neighbours are likely to be very connected as well. Conversely, a node with a small core index should have fewer neighbours, hence reducing the variety of the context.

Thus, instead of running a fixed number of random walks for every vertex of the graph (as it is done with DeepWalk and Node2Vec), we propose to scale this number according to the k-core index, running fewer walks for nodes with a low core index. To be more specific, we propose to scale the number of random walks rooted in a given node v in the graph as follows:

$$n_v = \max(\lfloor n \times \frac{k_v}{k_{degeneracy}} \rfloor, 1) \tag{13}$$

Where n_v is the number of random walks rooted in v to generate, k_v is the core index of node v, $k_{degeneracy}$ the degeneracy of the graph, n an arbitrary integer representing the maximum number of random walks rooted in v to generate (reached when v is in the $k_{degeneracy}$-core of the graph), and $\lfloor . \rfloor$ represents the integer flooring operator.

In most of the graphs, the number of nodes per core decreases with the core index, such that the low index cores have a lot of nodes and the high index cores have few to very few nodes. Applying this method reduces drastically the number

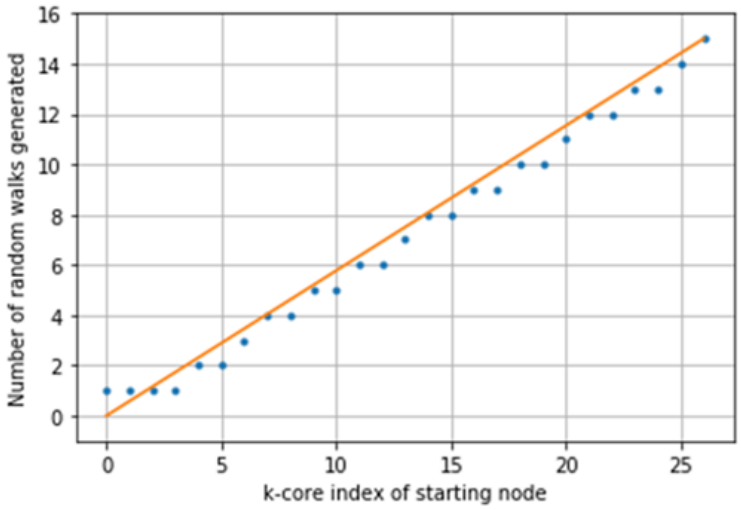


Figure 1: Number of random walks generated (blue dots) versus root core index - $n = 15$ and $k_{degeneracy} = 26$

of random walks to be run for the whole graph, as most of the vertices belong to cores where few random walks are drawn. Even if the task of drawing the random walks can be distributed, and so can be done efficiently, this process of scaling the number of walks still leads to a significant gain in execution time.

We should also notice that the set of all the random walks constitutes the training set of the SkipGram model. With the presented treatment, it get significantly reduced, inducing a way faster training process. However, because the training dataset is smaller, the results could tend to be less accurate. In other words, the quality of the final embedding could be degraded. As stated before, the main goal of this work is to limit this effect as much as possible. We justify our framework with the assumption that running more walks on nodes with simple context would simply add redundant information, and so not running them does not reduce much quality of the embeddings. Additionally, if the quality of the embedding becomes too low, we can use the scaling rule of the walks to increase the size of the dataset. The scaling rule can be used as a parameter to reach a target precision loss in the embedding, compared to a baseline with no reduction of walks.

2.2 Mean embedding propagation

Following the idea of MILE and HARP presented earlier, we propose a framework based on the work presented in [23] by *G. Salha et al*. The coarsening procedure we used simply consists in reducing the graph by using the core degeneracy. As the k-core subgraphs (also called k-degenerates) form a decreasing sequence of graphs (with respect to the inclusion relationship described in 2), we obtain a usable sequence of "coarsened" graphs. Then we use an embedding algorithm to embed the k_0 core subgraph (for a given k_0 index, smaller than the maximum k index). For instance, we can use DeepWalk or our algorithm with core adaptive random walks. Then, we use the propagation framework from [23] to spread the embeddings obtained at k_0-core to the whole graph.

Their idea is quite simple and shows quality results. Consider that we have built embeddings for the k-degenerate graph, and we want to propagate them to the $k-1$-degenerate graph (which is a supergraph of the k-degenerate one). Then, each node to embed will be assigned an embedding that is the average of all its neighbors that are either already embedded (from the k-degenerate graph) or about to be embedded (from the $k-1$-degenerate graph, but not in the $k-2$-degenerate). We obtain as much equations as there are nodes to be embedded, and as much unknown variables. The new propagated embeddings are the solutions of such a system. To avoid exactly solving the system, *G. Salha et al* introduced an approximation iterative calculus (more information can be found in their paper), which is executed in linear time with respect to the shape of the system (instead of cubic time for exact solution). Eventually, step by step, we propagate up to the whole graph.

To summarize, decomposing the graph into its k-core subgraphs is a fast process, even for large graphs. The k_0-degenerate subgraph needs to be significantly smaller than the original graph, which is usually true, for k_0 large enough. Thus, the embedding step is executed way faster on such subgraph than on the whole graph. Finally, the propagation phase consists in k_0 linear steps (with respect to the size of the successive layers of nodes), which is also really fast (compared to the embedding step). The process leads to a significant gain of time, however the propagation step can

have consequences on the quality of the final embeddings. To mitigate this impact, one can reduce the value of k_0 (the initial degenerate graph to embed with a time expensive strategy). Indeed, with lower k_0, the subgraph is larger, and there are less propagation steps, but the embedding step becomes longer. Our contribution has been to use this framework with a random walk based embedding method (DeepWalk).

3 Experiments

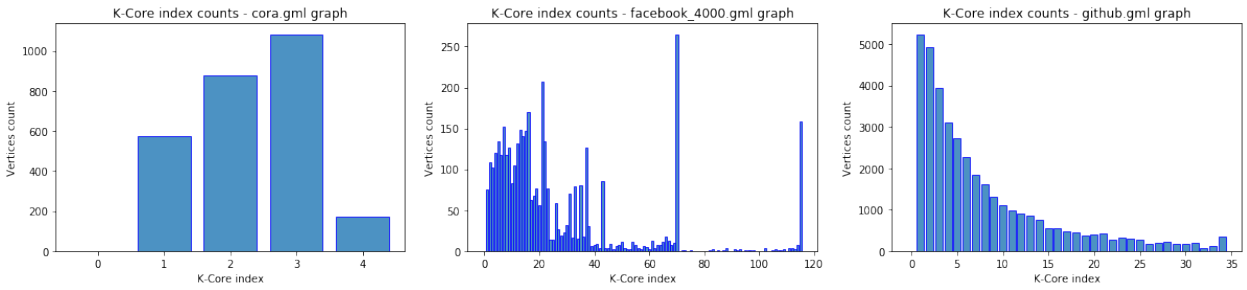
Let us describe our experimental settings before presenting our results.

3.1 Experimental setting

3.1.1 Dataset

Experiments presented in this paper were performed using three graphs of increasing size: Cora [1] ($2,708$ nodes, $5,429$ edges) and two graphs from Stanford's SNAP project [2]: a subgraph from Facebook ($4,039$ nodes, $88,234$ edges) and a Github developers' connections graph ($37,700$ nodes, $289,003$ edges). These are considered unweighted and undirected.

We plotted the distribution of nodes amongst the different k-degenerate subgraphs. Note that $(k+1)$-degenerate subgraph is included within k-degenerate, so we plotted the number of nodes that only belong to k-degenerate without belonging to $(k+1)$-degenerate, such that each vertex is counted once.



(a) Cora (b) Facebook subgraph (c) Github

Figure 2: Number of nodes with respect to core index

We should notice that Github graph is quite "regular". The cores nodes distribution follows a behavior that we could reasonably expect : numerous vertices in lowest core indices, decreasing number of nodes with respect to core index, very few in the highest core indices. Cora does not follow the same behavior at all. The graph is quite erratic, with a lot of pairs. Facebook graph is also a bit irregular. It follows the same tendency as Github graph, but the node distribution presents some spikes that are quite unexpected, especially around the 70-core and the last one.

3.1.2 Task

The model is evaluated on a *link prediction* task. For this, one first randomly removes a portion of the edges of the graph (10%, 30%, and 50% in our experiments), and trains the model on the resulting graph to obtain the embeddings. Training, validation and test set are obtained using those remove edges as positive samples, and the same number of unconnected pairs of nodes is randomly sampled to account for negative samples. A logistic regression is trained using the concatenation of the embeddings of two nodes as input, predicting the probability of these nodes being connected.

Performance of the embedding is evaluated using the *F1-score* of the downstream prediction task. Each experiment was repeated 5 times, and the standard deviations are reported along with the results.

[1] https://relational.fit.cvut.cz/dataset/CORA

[2] http://snap.stanford.edu/data/index.html

Experiments were run using two different CPUs, a *Intel Core i5, 2.4 GHz* with *8 GB* of RAM for the Cora and Facebook networks, and a *Intel Xeon E5-2630 v3, 2.4GHz* with *64 GB* of RAM for the Github graph. In the implementation, most graphs operations were performed using *networkx* [3], and propagation using *sparse scipy* [4].

Additional experiments were performed on the node prediction task, i.e. using the embedding to predict the label of a node for labelled graphs. The model did not provide satisfactory results, suggesting this task requires finer information to perform accurately.

Main results will be presented in this section, but complete experimental results can be found in the Appendix [4]. Since the focus of this paper is to study degeneracy-based models to improve random-walk-based embeddings, DeepWalk [11] will be used as baseline throughout the experiments.

All DeepWalk parameters are kept to default values, that is 15 random walks per node, of length 30, with a window size of 4. The dimension of the results embedding is 150.

3.2 Results

3.2.1 Small-size graphs

For Cora, we apply the propagation framework using DeepWalk as base embedder, on the 2-core up to the k-degenerate, that is the 3-core for the graph after a portion of edges are removed. Experimental results can be found in Table 1, for *link prediction* with 10% of edges removed. As mentioned before, one would expect the performance to decrease when the initial embedding is performed on a k_0-core with increasing k_0. Indeed, if the k_0 increases, the size of the k_0-degenerate graph decreases, making it more difficult for the random-walk based embedder to capture meaningful information, and increasing the propagation task complexity since there are more nodes to propagate the embeddings to.

Model	**Performances** (%)		**Execution time** (sec.)	
	F1 - Score	Perf. Drop w.r.t. Baseline	**Total**	*Speedup*
DeepWalk	58.35 ($\pm$ 1.35)		37.45 ($\pm$ 2.37)	
2-core (Dw)	58.45 ($\pm$ 0.37)	0.2	29.77 ($\pm$ 1.62)	x1.3
3-core (Dw)	**59.21** ($\pm$ 0.9)	**1.5**	**14.05** ($\pm$ 0.58)	**x2.7**

Table 1: *Link prediction* on Cora graph, with 10% of edges removed. Comparison of the DeepWalk baseline with our propagation framework, with DeepWalk as base embedder (K-core(Dw)).

Surprisingly, on average, this expected performance drop is not observed. However, the standard deviation in each experiment is high, and the overall *F1-score* rather low, making it difficult to draw any conclusion. Yet, a slight speedup of x2.7 is obtained at a reasonable cost.

3.2.2 Medium/Large-size graphs

Similar experiment was performed on the Facebook graph, using DeepWalk to embed the initial k_0-core, and performing *link prediction* with increasing values of k_0. Results summary can be found in Table 2. As expected, as k_0 increases, the $F1$-score decreases, but this drop is at most 11.9% compared to the DeepWalk baseline. On the other side, a significant speedup of x14.6 is achieved on the highest core.

The Core-Adaptive Random walk method (Corewalk) introduced in section 2.1 was also tested using this graph and the *link prediction* with 10% missing edges. A summary of the results are reported in table 3. Alone, without any propagation technique, Corewalk already shows promising results : the $F1$-score obtained is 2% higher than Deepwalk baseline, while having a speedup of x3. Consistent results are obtain when removing 30%, demonstrating robustness of the solution.

As k_0 increases, one observe similar performance drop, at most 11.4% lower than Deepwalk, and the speedup reaches x13.9.

These two experiments are summarized in Figure 3 and Figure 4, for *link prediction* with 10% and 30% removed edges respectively. First, one can see that the pattern obtained on the 10% task when reaching high core index, *i.e* the increase of *F1-score* with high uncertainty, do not generalize. This pattern is thus most probably due to the particular structure

[3] https://networkx.github.io

[4] https://docs.scipy.org/doc/scipy/reference/sparse.html

Model	**Performances** (%)		**Execution time** (sec.)	
	F1 - Score	Perf. Drop w.r.t Baseline	**Total**	*Speedup*
DeepWalk	**71.67** ($\pm$ 0.33)		101.92 ($\pm$ 2.93)	
9-core (Dw)	69.31 ($\pm$ 0.32)	**-3.3**	70.51 ($\pm$ 0.12)	x1.4
25-core (Dw)	67.53 ($\pm$ 0.49)	-5.8	48.91 ($\pm$ 3.56)	x2.1
49-core (Dw)	63.16 ($\pm$ 0.36)	-11.9	21.74 ($\pm$ 1.07)	x4.7
73-core (Dw)	66.14 ($\pm$ 2.28)	-7.7	7.89 ($\pm$ 0.04)	x12.9
97-core (Dw)	66.57 ($\pm$ 0.7)	-7.1	**7** ($\pm$ 0.34)	**x14.6**

Table 2: *Link prediction* on Facebook graph, with 10% of edges removed. Comparison of the DeepWalk baseline with our propagation framework, with DeepWalk as base embedder (K-core(Dw)).

Model	**Performances** (%)		**Execution time** (sec.)	
	F1 - Score	Perf. Drop w.r.t. Baseline	**Total**	*Speedup*
CoreWalk	**73.07** ($\pm$ 0.25)	**2**	33.5 ($\pm$ 0.53)	x3
9-core (Cw)	68.84 ($\pm$ 0.26)	-4	32.97 ($\pm$ 0.86)	x3.1
25-core (Cw)	68.31 ($\pm$ 0.3)	-4.7	23.98 ($\pm$ 1.01)	x4.3
49-core (Cw)	63.51 ($\pm$ 0.48)	-11.4	16.88 ($\pm$ 0.73)	x6
73-core (Cw)	65.55 ($\pm$ 2.77)	-8.5	8.64 ($\pm$ 0.53)	x11.8
97-core (Cw)	66.56 ($\pm$ 0.62)	-7.1	**7.34** ($\pm$ 0.95)	**x13.9**

Table 3: *Link Prediction* on Facebook graph, with 10% of edges removed. Comparison of the DeepWalk baseline with the Core-Adaptive Random Walks (CoreWalk) and our propagation framework, with CoreWalk as base embedder. (K-core(Cw))

of the graph (*i.e.* important spike around core 60). Then, it is noticeable that the gain in performance using Corewalk do not hold when combined with the propagation technique for high k_0. However, one can see that the execution time remains lower until both execution time converge, when the number of nodes to embed initially becomes very low.

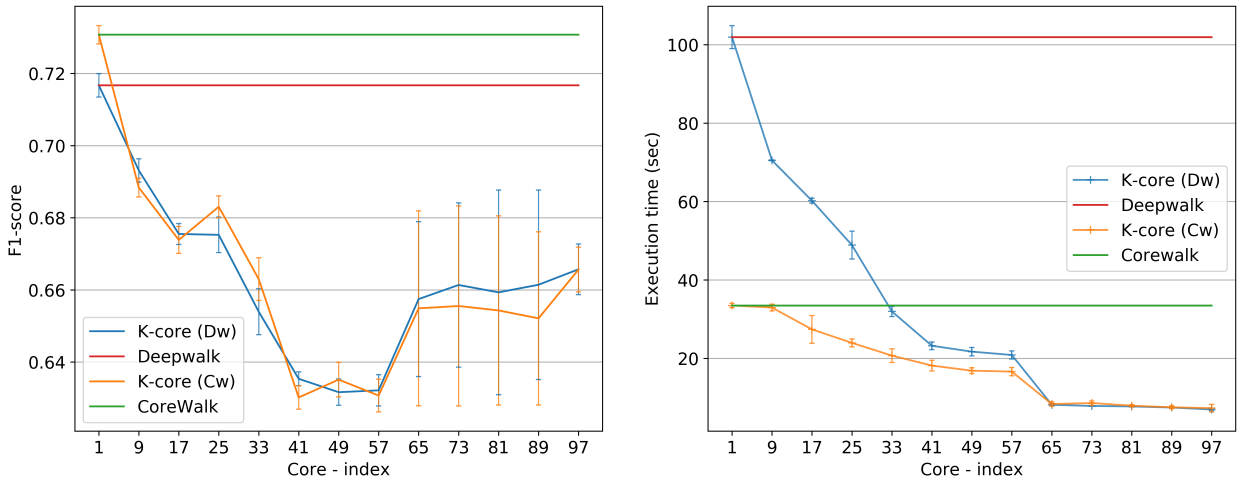


Figure 3: Experiment summary of the *link prediction* task, with 10% of edges removed. (Left) $F1 - score$ as a function of the initial embedded core index. (Right) Total execution time as a function of the initial embedded core index.

Details of this execution time can be found in Figure 5. The execution time is always dominated by the embedding time of the k_0-core, ranging from 100 to 6 seconds, when both core decomposition time and propagation time remains below 1 second. This embedding time is directly proportional to the number of nodes in the corresponding k_0-core to embed.

A final experiment was performed on a Github graph, around 10 times bigger than Facebook, to attest the scalability of the framework. From results displayed in Table 4, one can notice the sudden drop in performance (-10.6%), even when the initial k_0 is rather large (10-core), with a slight speedup (x3.2). However, the performance drop lessens for

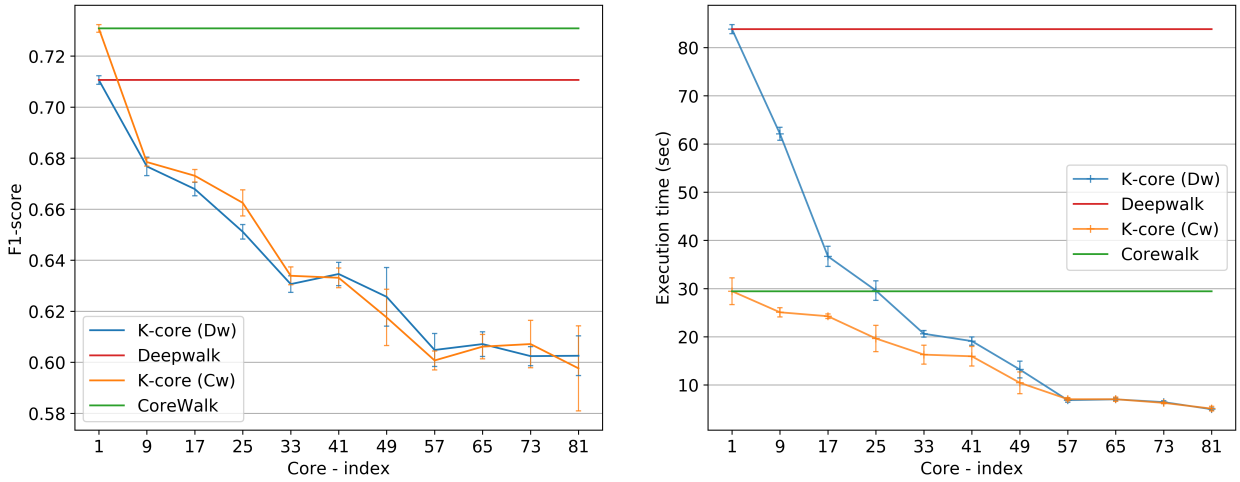


Figure 4: Experiment summary of the *link prediction* task, with 30% of edges removed. (Left) $F1 - score$ as a function of the initial embedded core index. (Right) Total execution time as a function of the initial embedded core index.

higher core, providing satisfactory results, with around 15% drop in $F1$-score, and an execution time going from more than 10 minutes to around 30 seconds.

Model	**Performances** (%)		**Execution time** (sec.)	
	F1 - Score	Perf. Drop w.r.t. Baseline	**Total**	*Speedup*
DeepWalk	**74.53** ($\pm$ 0.04)		684.15 ($\pm$ 4.87)	
10-core (Dw)	66.71 ($\pm$ 0.2)	**-10.5**	214.41 ($\pm$ 1.71)	x3.2
20-core (Dw)	64.72 ($\pm$ 0.08)	-13.2	84.25 ($\pm$ 2.02)	x8.1
30-core (Dw)	63.76 ($\pm$ 0.43)	-14.4	**33.19**($\pm$ 0.99)	**x20.6**

Table 4: *Link prediction* on Github graph, with 10% of edges removed. Comparison of the DeepWalk baseline with our propagation framework, with DeepWalk as base embedder (K-core(Dw)).

3.2.3 Embedding visualization

Visualization of the embeddings are provided in the Appendix, in section 4.3. They are obtained by using Principal Component Analysis (PCA) as dimensional reduction method to project the embeddings onto a 2D space. Two situations are presented :

- The most common case, in Figure 6, where the initial embedding is performed on a connected graph. On may notice (6b), that the framework tends to shrink the point cloud to its center in the original projected space of the PCA at each propagation step. The appearing lines in Figure 6a suggests that the framework creates embeddings with very low variance in most dimensions.
- Figure 7 depicts the case when the initial embedding is performed on a k_0-core not connected. DeepWalk will then create two distant point clouds, as appearing in 7b. The propagation will then put most of the variance in the embedding in this direction to link the two point clouds. This is a drawback of the propagation technique, since the relative positions of the two original point clouds are not related to any property of the whole graph.

4 Conclusion and Discussion

Let us recall our guiding principle : accelerate embedding techniques taking advantage of core degeneracy with a limited loss of accuracy. We used downstream tasks to build a metric over the quality of the embeddings produced and DeepWalk as a baseline to compare. A drawback of this approach is that the measure of the embedding quality is biased towards this specific task. To ensure the robustness of the proposed approaches, further work should diversify

both the data used (graphs) and the downstream tasks to execute. We still produced promising results that could be an interesting starting step for further work.

Based on random walks algorithms, we proposed a core adaptive framework that scales the number of random walks to explore the nodes' contexts. We observed significant gains of running time on every test graph, with a really moderate drop in prediction tasks. We even observed a slight improvement for larger graphs. Additionally, we reproduced a propagation framework proposed by *G. Salha et al* [23]. We used it with DeepWalk and our Core Adaptive Random Walks framework as embedders instead of auto encoders (as they do in their paper). It resulted in an even faster embedding process. Choosing the k_0 core index for initial embeddings allows some flexibility to reach a satisfactory loss in performance.

However, the embeddings obtained with such random walks based methods did not lead to outstanding accuracy scores. Indeed, they build graph representations simply using graphs' structural properties. They did not take into account nodes labels or attributes. Thus, our baselines scores are quite low, compared to what can be done with current state of the art techniques (see Graph Convolutional Networks with Attention[17] or Graph Attenuated Attention Networks[24]). DeepWalk was published in 2014, Node2Vec in 2016. To make sure that our frameworks are viable and robust, they should be challenged with more efficient baseline methods as there would be more margin for score loss.

One thing that further work should focus on is connectivity. It is a notion strongly related to the problems we faced during our work, and we believe that improvements can be achieved in this direction. It might happen at some point of the k-core decomposition that the obtained subgraph is not connected anymore (imagine a connected graph with two dense areas far one from the other). Then the embedding step creates two clusters of point in the representation space. The mean propagation framework succeeds in connecting them by placing points in between (see figure 7b), but the final embeddings are of quite poor quality and do not match well the structure of the graph. An idea to tackle this problem is still to be explored, for instance, we could think of a way to relate not-connected embeddings via generating random walks between the connected areas.

Another way of improvement would be to refine the propagation of the embeddings. In the propagation step, as the index of the k-core decreases, a large number of nodes might be added, compared to the number of nodes that have already been assigned an embedding. Thus, we might not have enough information to precisely build nodes representations, and the averaging step could crush together the points. An idea could be to do mean propagation if few nodes are added, and to recompute embeddings if the nodes are too numerous. However, it would be necessary to find a way to compute new embeddings using the ones we already have.

References

[1] J Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Advances in neural information processing systems*, pages 41–50, 2006.

[2] Giannis Nikolentzos, Polykarpos Meladianos, Stratis Limnios, and Michalis Vazirgiannis. A degeneracy framework for graph similarity. In *IJCAI*, pages 2595–2601, 2018.

[3] Christos Giatsidis, Fragkiskos Malliaros, Dimitrios Thilikos, and Michalis Vazirgiannis. Corecluster: A degeneracy based graph clustering framework. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[4] Reid Andersen, Fan Chung, and Kevin Lang. Local graph partitioning using pagerank vectors. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 475–486. IEEE, 2006.

[5] Francois Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on knowledge and data engineering*, 19(3):355–369, 2007.

[6] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[7] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000.

[8] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in neural information processing systems*, pages 585–591, 2002.

[9] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Grarep: Learning graph representations with global structural information. In *Proceedings of the 24th ACM international on conference on information and knowledge management*, pages 891–900, 2015.

[10] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. Asymmetric transitivity preserving graph embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1105–1114, 2016.

[11] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '14*, 2014.

[12] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[13] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks, 2016.

[14] Daixin Wang, Peng Cui, and Wenwu Zhu. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1225–1234, 2016.

[15] Shaosheng Cao, Wei Lu, and Qiongkai Xu. Deep neural networks for learning graph representations. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[16] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[17] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[18] Thomas N Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

[19] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.

[20] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[21] Haochen Chen, Bryan Perozzi, Yifan Hu, and Steven Skiena. Harp: Hierarchical representation learning for networks, 2017.

[22] Jiongqian Liang, Saket Gurukar, and Srinivasan Parthasarathy. Mile: A multi-level framework for scalable graph embedding, 2018.

[23] Guillaume Salha, Romain Hennequin, Viet Anh Tran, and Michalis Vazirgiannis. A degeneracy framework for scalable graph autoencoders, 2019.

[24] Rui Wang, Bicheng Li, Shengwei Hu, Wenqian Du, and Min Zhang. Knowledge graph embedding via graph attenuated attention networks. *IEEE Access*, 2019.

Appendix

4.1 Experimental results

Model	**Performances**(%)		**Execution time** (sec.)				
	F1 - Score	Perf. Drop w.r.t Baseline	Core decomposition	Propagation	Embedding	**Total**	*Speedup*
DeepWalk	58.35 (± 1.35)					37.45 (± 2.37)	
2-core (Dw)	58.45 (± 0.37)	0.2	0.05	**0.28**	29.44	29.77 (± 1.62)	x1.3
3-core (Dw)	**59.21** (± 0.9)	**1.5**	**0.03**	**0.28**	**13.74**	**14.05** (± 0.58)	**x2.7**

Table 5: *Link prediction* on Cora graph, with 10% of edges removed. Comparison of the DeepWalk baseline with our propagation framework, with DeepWalk as base embedder. (K-core(Dw))

Model	**Performances** (%)		**Execution time** (sec.)				
	F1 - Score	Perf. Drop w.r.t Baseline	Core decomposition	Propagation	Embedding	**Total**	*Speedup*
DeepWalk	60.32 (± 0.51)					32.96 (± 0.81)	
2-core (Dw)	**60.41** (± 0.94)	**0.1**	0.07	**0.24**	23.47	23.78 (± 1.09)	x1.4
3-core (Dw)	59.71 (± 0.6)	-1	**0.02**	0.39	**5.91**	**6.32** (± 0.75)	**x5.2**

Table 6: *Link prediction* on Cora graph, with 30% of edges removed. Comparison of the DeepWalk baseline with our propagation framework, with DeeWalk as base embedder. (K-core(Dw))

Model	**Performances** (%)		**Execution time** (sec.)				
	F1 - Score	Perf. Drop w.r.t Baseline	Core decomposition	Propagation	Embedding	**Total**	*Speedup*
DeepWalk	71.67 (± 0.33)					101.92 (± 2.93)	
9-core (Dw)	69.31 (± 0.32)	-3.3	0.6	**0.56**	69.35	70.51 (± 0.12)	x1.4
17-core (Dw)	67.55 (± 0.29)	-5.7	0.58	0.75	58.89	60.21 (± 0.6)	x1.7
25-core (Dw)	67.53 (± 0.49)	-5.8	0.48	0.92	47.5	48.91 (± 3.56)	x2.1
33-core (Dw)	65.39 (± 0.64)	-8.8	0.41	0.79	30.79	31.99 (± 1.29)	x3.2
41-core (Dw)	63.54 (± 0.19)	-11.4	0.33	0.86	22.05	23.24 (± 0.98)	x4.4
49-core (Dw)	63.16 (± 0.36)	-11.9	0.32	0.91	20.5	21.74 (± 1.07)	x4.7
57-core (Dw)	63.22 (± 0.43)	-11.8	0.31	0.9	19.67	20.89 (± 1.05)	x4.9
65-core (Dw)	65.75 (± 2.15)	-8.3	0.19	0.93	7.07	8.2 (± 0.33)	x12.4
73-core (Dw)	66.14 (± 2.28)	-7.7	0.19	0.92	6.78	7.89 (± 0.04)	x12.9
81-core (Dw)	65.93 (± 2.84)	-8	0.19	0.94	6.64	7.77 (± 0.15)	x13.1
89-core (Dw)	66.14 (± 2.63)	-7.7	**0.18**	0.96	6.35	7.49 (± 0.17)	x13.6
97-core (Dw)	66.57 (± 0.7)	-7.1	**0.18**	0.93	**5.89**	**7** (± 0.34)	**x14.6**
CoreWalk	**73.07** (± 0.25)	**2**	0.65		32.85	33.5 (± 0.53)	x3
9-core (Cw)	68.84 (± 0.26)	-4	0.62	0.62	31.73	32.97 (± 0.86)	x3.1
17-core (Cw)	67.39 (± 0.37)	-6	0.61	0.76	26.05	27.41 (± 3.53)	x3.7
25-core (Cw)	68.31 (± 0.3)	-4.7	0.45	0.82	22.7	23.98 (± 1.01)	x4.3
33-core (Cw)	66.3 (± 0.59)	-7.5	0.37	0.9	19.45	20.73 (± 1.73)	x4.9
41-core (Cw)	63.02 (± 0.33)	-12.1	0.33	1.02	16.84	18.19 (± 1.35)	x5.6
49-core (Cw)	63.51 (± 0.48)	-11.4	0.32	1.08	15.48	16.88 (± 0.73)	x6
57-core (Cw)	63.07 (± 0.45)	-12	0.31	1.1	15.24	16.65 (± 1.06)	x6.1
65-core (Cw)	65.49 (± 2.7)	-8.6	0.2	1.03	7.16	8.39 (± 0.54)	x12.2
73-core (Cw)	65.55 (± 2.77)	-8.5	0.2	1.21	7.22	8.64 (± 0.53)	x11.8
81-core (Cw)	65.43 (± 2.62)	-8.7	0.19	1.05	6.73	7.96 (± 0.26)	x12.8
89-core (Cw)	65.21 (± 2.4)	-9	**0.18**	1.01	6.35	7.55 (± 0.37)	x13.5
97-core (Cw)	66.56 (± 0.62)	-7.1	**0.18**	1.04	6.11	7.34 (± 0.95)	x13.9

Table 7: *Link prediction* on Facebook graph, with 10% of edges removed. Comparison of the DeepWalk baseline with the Core-Adaptive Random Walks (CoreWalk) and our propagation framework, with DeepWalk and CoreWalk as base embedder. (K-core(Dw) and K-core(Cw) respectively)

Model	**Performances** (%)		**Execution time** (sec.)				
	F1 - Score	Perf. Drop w.r.t Baseline	Core decomposition time	Propagation time	Embedding time	**Total**	*Speedup*
DeepWalk	71.06 (± 0.17)					83.83 (± 0.95)	
9-core (Dw)	67.68 (± 0.36)	-4.8	0.44	**0.6**	61.06	62.1 (± 1.35)	x1.3
17-core (Dw)	66.79 (± 0.27)	-6	0.37	0.74	35.57	36.68 (± 2.09)	x2.3
25-core (Dw)	65.12 (± 0.29)	-8.4	0.32	0.76	28.51	29.59 (± 2.03)	x2.8
33-core (Dw)	63.06 (± 0.32)	-11.3	0.27	0.79	19.57	20.62 (± 0.67)	x4.1
41-core (Dw)	63.46 (± 0.46)	-10.7	0.25	0.82	18.04	19.11 (± 0.86)	x4.4
49-core (Dw)	62.57 (± 1.15)	-12	0.2	0.81	12.2	13.21 (± 1.73)	x6.3
57-core (Dw)	60.48 (± 0.65)	-14.9	0.14	0.93	5.77	6.84 (± 0.43)	x12.3
65-core (Dw)	60.71 (± 0.48)	-14.6	0.14	0.89	5.98	7.02 (± 0.27)	x11.9
73-core (Dw)	60.24 (± 0.37)	-15.2	0.14	0.88	5.42	6.44 (± 0.28)	x13
81-core (Dw)	60.26 (± 0.78)	-15.2	**0.13**	0.83	**4.01**	**4.96** (± 0.27)	**x16.9**
CoreWalk	**73.08** (± 0.15)	**2.8**	0.5		28.73	29.45 (± 2.76)	x2.8
9-core (Cw)	67.85 (± 0.16)	-4.5	0.45	0.64	23.99	25.08 (± 0.95)	x3.3
17-core (Cw)	67.31 (± 0.24)	-5.3	0.39	0.76	23.12	24.26 (± 0.46)	x3.5
25-core (Cw)	66.25 (± 0.51)	-6.8	0.32	0.88	18.44	19.64 (± 2.71)	x4.3
33-core (Cw)	63.39 (± 0.35)	-10.8	0.26	0.97	15.06	16.29 (± 1.97)	x5.1
41-core (Cw)	63.31 (± 0.39)	-10.9	0.25	1	14.69	15.95 (± 2.03)	x5.3
49-core (Cw)	61.76 (± 1.1)	-13.1	0.19	0.91	9.35	10.45 (± 2.25)	x8
57-core (Cw)	60.07 (± 0.37)	-15.5	0.15	0.87	6.03	7.05 (± 0.31)	x11.9
65-core (Cw)	60.61 (± 0.48)	-14.7	0.15	0.97	5.93	7.04 (± 0.43)	x11.9
73-core (Cw)	60.72 (± 0.93)	-14.6	0.15	0.87	5.26	6.28 (± 0.16)	x13.4
81-core (Cw)	59.76 (± 1.67)	-15.9	**0.13**	0.84	4.13	5.09 (± 0.46)	x16.5

Table 8: *Link prediction* on Facebook graph, with 30% of edges removed. Comparison of the DeepWalk baseline with the Core-Adaptive Random Walks (CoreWalk) and our propagation framework, with DeepWalk and CoreWalk as base embedder. (K-core(Dw) and K-core(Cw) respectively)

Model	**Performances** (%)		**Execution time** (sec.)				
	F1 - Score	Perf. Drop w.r.t Baseline	Core decomposition	Propagation	Embedding	**Total**	*Speedup*
DeepWalk	**74.53** (± 0.04)					684.15 (± 4.87)	
10-core (Dw)	66.71 (± 0.2)	**-10.5**	2.85	**18.45**	193.11	214.41 (± 1.71)	x3.2
20-core (Dw)	64.72 (± 0.08)	-13.2	2.23	21.07	60.95	84.25 (± 2.02)	x8.1
30-core (Dw)	63.76 (± 0.43)	-14.4	**1.84**	20.58	**10.77**	**33.19** (± 0.99)	**x20.6**

Table 9: *Link prediction* on Github graph, with 10% of edges removed. Comparison of the DeepWalk baseline with our propagation framework, with DeepWalk as base embedder. (K-core(Dw))

Model	**Performances** (%)		**Execution time** (sec.)				
	F1 - Score	Perf. Drop w.r.t Baseline	Core decomposition	Propagation	Embedding	**Total**	*Speedup*
DeepWalk	**71.92** (± 0.17)					631.97 (± 2.96)	
10-core (Dw)	65.27 (± 0.26)	**-9.2**	1.92	**17.55**	137.77	157.24 (± 3.4)	x4
20-core (Dw)	64.45 (± 0.28)	-10.4	**1.42**	18.9	**27.65**	**47.97** (± 0.39)	**x13.2**

Table 10: *Link prediction* on Github graph, with 30% of edges removed. Comparison of the DeepWalk baseline with our propagation framework, with DeepWalk as base embedder. (K-core(Dw))

4.2 Detailed execution time

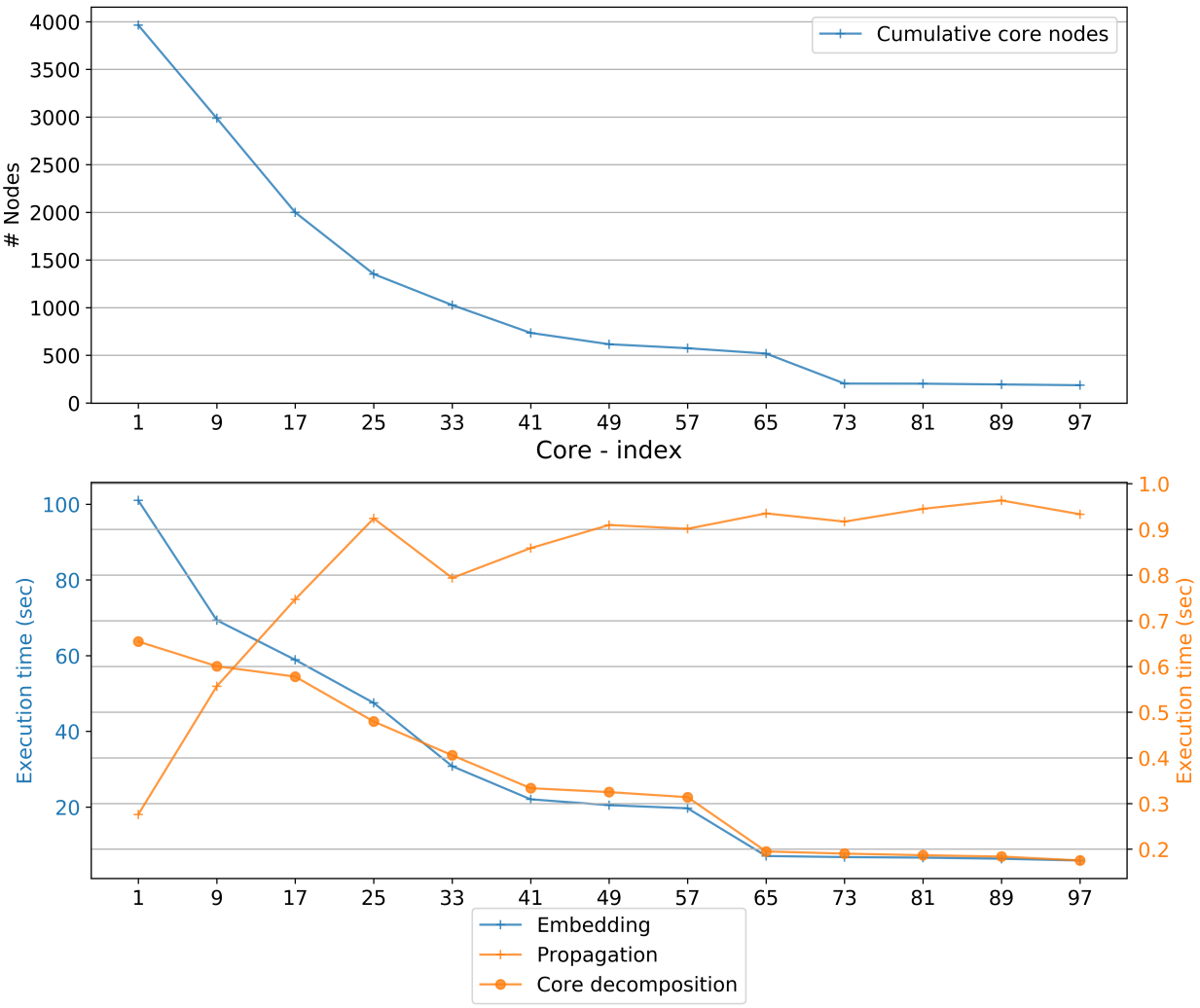
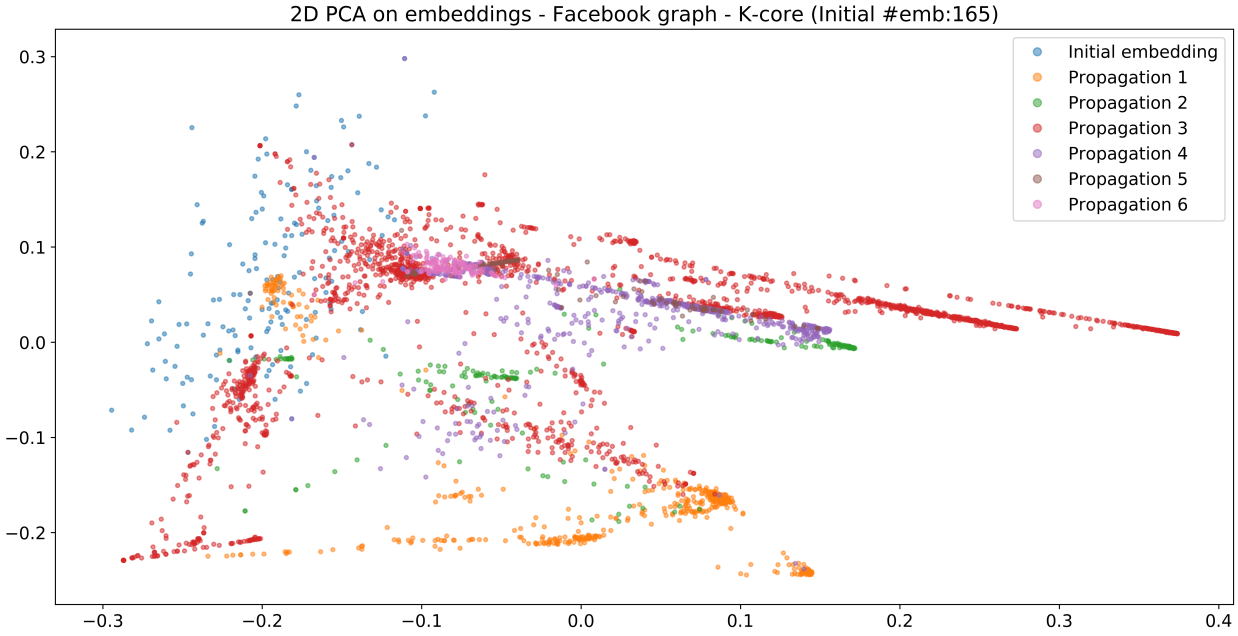
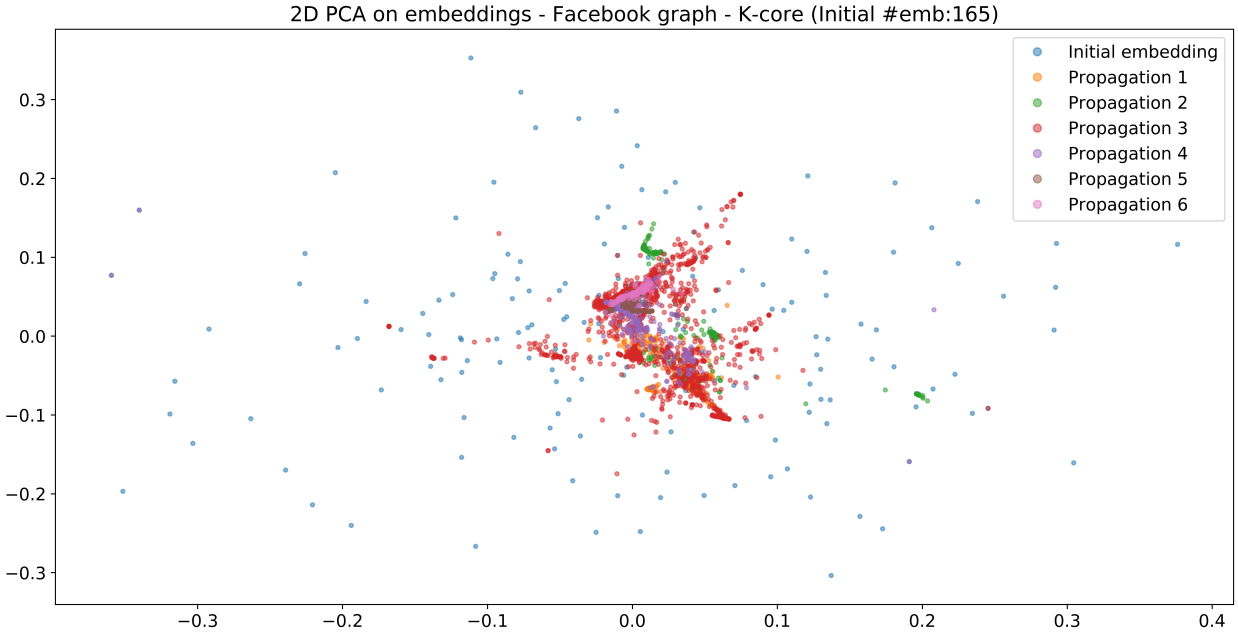


Figure 5: Execution time summary for the *link prediction* task on Facebook graph, 10% of edges removed. (Top) Number of nodes in the initial k-core to embed. (Bottom) Execution time details as a function of the initial k-core index.

4.3 Embedding visualization

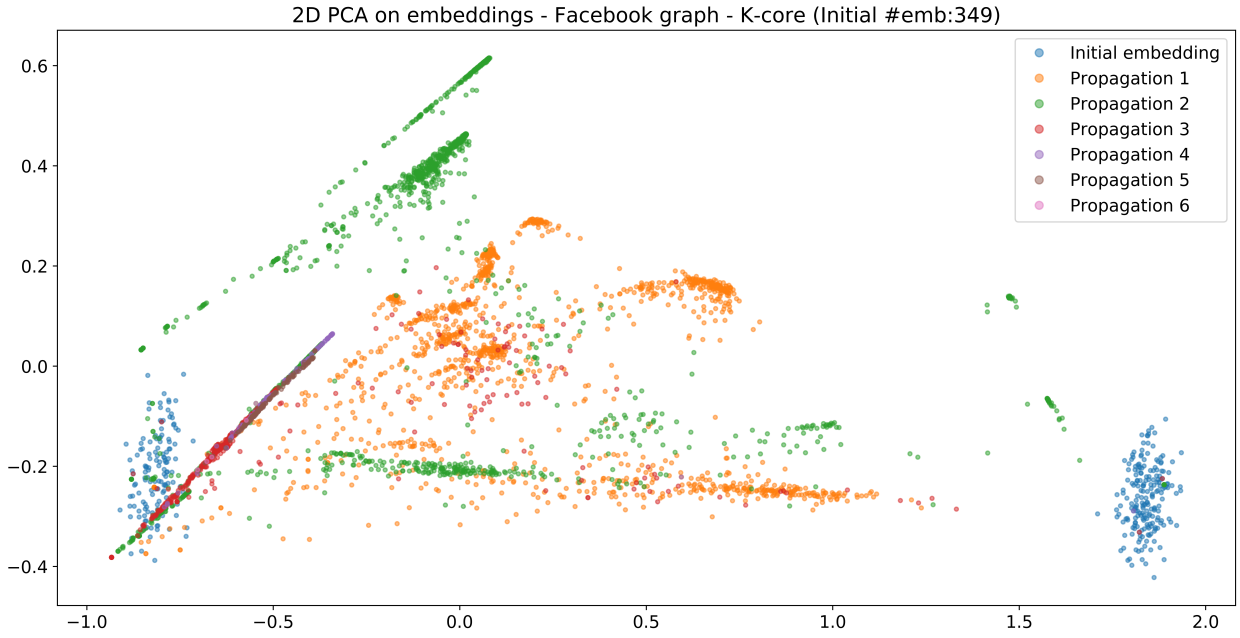


(a) PCA fitted on the final embedding of the whole graph

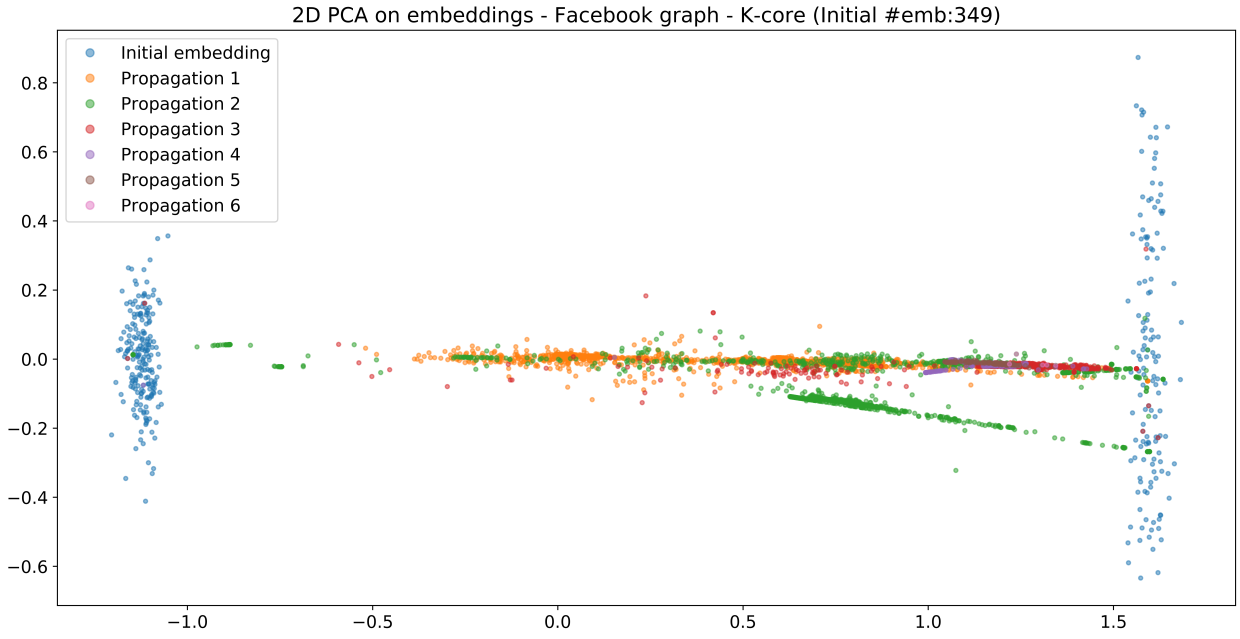


(b) PCA fitted to the embeddings of the initial 103-core

Figure 6: Visualization of the embeddings obtained on Facebook graph, with 10% of edges removed, and initial embedding performed on the connected 103-core



(a) PCA fitted on the final embedding of the whole graph



(b) PCA fitted to the embeddings of the initial 64-core

Figure 7: Visualization of the embeddings obtained on Facebook graph, with 10% of edges removed, and initial embedding performed on the not-connected 64-core